USENIX Association

Proceedings of the

2nd USENIX Windows NT Symposium

August 3-5, 1998
Seattle, Washington

Symposium Organizers

Program Co-Chairs
Susan Owicki, *InterTrust Technologies Corporation*
Thorsten von Eicken, *Cornell University*

Steering Committee
Ed Lazowska, *University of Washington*
Michael B. Jones, *Microsoft Research, Microsoft Corporation*
Margo Seltzer, *Harvard University*

Program Committee
John Bennett, *Rice University*
Pei Cao, *University of Wisconsin, Madison*
J. Bradley Chen, *Harvard University & Appliant Inc.*
Anton Chernoff, *Digital Equipment Corporation*
Andrew Chien, *University of Illinois*
Jim Gray, *Microsoft Bay Area Research Center*
Carl Hauser, *Xerox Palo Alto Research Center*
Michael B. Jones, *Microsoft Research, Microsoft Corporation*
David Korn, *AT&T Labs Research*
Ed Lazowska, *University of Washington*
Jane Liu, *University of Illinois at Urbana-Champaign*
Nick Vasilatos, *DE Shaw Financial Technology LP*
Werner Vogels, *Cornell University*
Stephen R. Walli, *Softway Systems, Inc.*
Rumi Zahir, *Intel Corporation*

Additional Reviewers
Drew Hess, *Intel Corporation*
Hong Wang, *Intel Corporation*

The USENIX Association Staff

Table of Contents

2nd USENIX Windows NT Symposium

August 3-4, 1998
Seattle, Washington

Monday, August 3

Performance
Session Chair: Ed Lazowska, University of Washington

From UNIX to NT to UNIX
Session Chair: Stephen Walli, Softway Systems, Inc.

Threads
Session Chair: Rumi Zahir, Intel Corporation

Tuesday, August 4

Mixing UNIX and NT
Session Chair: David Korn, AT&T Labs – Research

Networking and Distributed Systems
Session Chair: Werner Vogels, Cornell University

Real Time Scheduling
Session Chair: Jane Liu, University of Illinois

Demonstrations and Posters
Session Chair: John Bennett, Rice University

A Performance Study of Sequential I/O on Windows NT™ 4

Erik Riedel
Carnegie Mellon University, riedel@cmu.edu
Catharine van Ingen, Jim Gray
Microsoft Research, {vanIngen,Gray}@microsoft.com

Abstract

Large-scale database, data mining, and multimedia applications require large, sequential transfers and have bandwidth as a key requirement. This paper investigates the performance of reading and writing large sequential files using the Windows NT™ 4.0 File System. The study explores the performance of Intel Pentium Pro™ based memory and IO subsystems, including the processor bus, the PCI bus, the SCSI bus, the disk controllers, and the disk media in a typical server or high-end desktop system. We provide details of the overhead costs at each level of the system and examine a variety of the available tuning knobs. We show that NTFS out-of-the-box performance is quite good, but overheads for small requests can be quite high. The best performance is achieved by using large requests, bypassing the file system cache, spreading the data across many disks and controllers, and using deep-asynchronous requests. This combination allows us to reach or exceed the half-power point of all the individual hardware components.

1 Introduction

High-speed sequential access is important for bulk data operations typically found in utility, multimedia, data mining, and scientific applications. High-speed sequential IO is also an important factor in the startup of interactive applications. Minimizing IO overhead and maximizing bandwidth frees power to actually process the data.

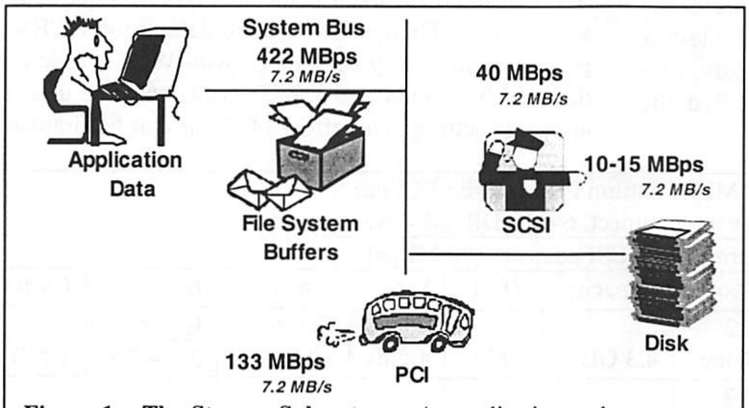


Figure 1 – The Storage Subsystem – An application makes requests of the file system, which transfers them across the PCI bus to a SCSI adapter that sends them across the SCSI bus to the disk. For each component, the upper numbers give the advertised speed and the lower number gives the actual speed in this application reading a single disk.

Figure 1 shows how data flows in a modern storage subsystem. Application requests are passed to the file system. If the file system cannot service the request from its main memory buffers, it passes requests to a host bus adapter (HBA) over a PCI peripheral bus. The HBA passes requests across the SCSI bus to the disk drive controller. The controller reads or writes the disk media and returns data via the reverse route.

The large, bold numbers in Figure 1 indicate the advertised throughputs listed on the boxes of the various hardware components. These are the figures quoted in hardware reviews and specifications. Several factors prevent you from achieving this PAP (peak advertised performance). The media-transfer speed and the processing power of the on-drive controller limit disk bandwidth. The wire's transfer rate, the disk transfer rate, and SCSI protocol overheads all limit throughput.

In the case diagrammed in Figure 1, the disk media is the bottleneck, limiting aggregate throughput to 7.2 MBps at each step of the pipeline. There is a significant gap between the advertised performance and this out-of-the-box performance. Moreover, the application consumes between 25% and 50% of the processor at this throughput. The processor would saturate long before it reached the advertised SCSI or PCI throughputs.

The goal of this study is to see if applications can do better cheaply - increase sequential IO throughput and decrease processor overhead while making as few application changes as possible. Our goal is to bring the real application performance (RAP) up to the *half-power point* - the point at which the system delivers at least half of the theoretical maximum performance. More succinctly, the goal is *RAP* $\geq$ *PAP/2*. Such improvements often represent significant (2x to 10x) gains over the out-of-the-box performance. We will see that the half-power point can be achieved without heroic effort, through a combination of techniques.

Our benchmark is a simple application that uses the NT file system to sequentially read and write a 100 MB file and times the result. **`ReadFileEx()`** and IO completion routines were used to keep *n* asynchronous requests in flight at all times. All measurements were repeated three times and, unless otherwise noted, all the data obtained were quite repeatable (within 3% margin of error). Multiple disk data was obtained by using NT *ftdisk* to build striped logical volumes and the basic system configuration used for all our measurements is described in Table 1.

The next section discusses our out-of-the-box measurements. Section 3 explores the basic capabilities of the hardware storage subsystem. Ways to improve performance by increasing parallelism are presented in Section 4. Section 5 provides more detailed discussion of performance limits and some additional software considerations. Finally, we summarize and suggest steps for additional study. An extended version of this paper, and all the benchmark software can be found at *www.research.microsoft.com/barc/Sequential_IO*.

2 Out-of-the-Box Performance

Our first measurements examine the out-of-the-box performance of our benchmark synchronously reading and writing using the NTFS defaults. The benchmark requests data sequentially from the file system. Since the data is not already in the file system cache, the file system fetches the data from disk into the cache and then copies it to the application's buffers. Similarly, when writing, the program's data is copied to the file cache and a separate thread asynchronously flushes the cache to disk in 64 KB units. In the out-of the-box experiments, the file being written was already allocated but not truncated. The program specified the **`FILE_FLAG_SEQUENTIAL_SCAN`** attribute when opening the file with **`CreateFile()`**. The total user and system processor time was measured via **`GetProcessTimes() and`** Figure 2 shows the results across a variety of application request sizes.

Buffered, sequential read throughput is nearly constant for request sizes up to 64 KB. The file system prefetches by issuing 64 KB requests to the disk. The disk controller also prefetches data from the media to its internal cache, which hides rotational delay and allows the disk to approach the media transfer limit. Figure 2 shows a sharp drop in read throughput for request sizes larger than 64 KB as the file system and disk prefetch mechanism fails (this problem is fixed in NT5). Figure 2 also indicates that buffered-sequential writes are substantially slower than reads. The file system performs write-back caching by default; it copies the contents of the application buffer into one or more file system buffers and the application considers the write complete when the copy is made. The file system then coalesces sequential requests into large 64 KB writes, leading to relatively constant throughput above 4 KB.

Disk controllers also implement write-through and write-back caching, controlled by the Write-Cache-Enable (WCE) setting directly at the device [SCSI93]. If WCE is disabled, the disk controller announces IO completion only after the media write is complete. If WCE is enabled, the disk announces write completion as soon as the data is stored in its cache and before the actual write onto the magnetic disk media. WCE allows the disk to hide the seek and media transfer, analogous to prefetching for reads. This improves write performance by giving pipeline parallelism – the write of the media overlaps the transfer of the next write on the SCSI even if the file system requests are synchronous. There is no standard default for WCE – a particular drive may be shipped with WCE enabled or disabled by default and a SCSI utility must be used to alter this setting. The effect of WCE can be dramatic

Processor	Gateway 2000 G6-200, 200 MHz Pentium Pro, 1 32-bit PCI bus 64-bit wide 66 MHz memory interconnect, 64MB DRAM 4-way interleave							
Host bus adapter	1 or 2 Adaptec 2940UW Ultra-Wide SCSI adapters (40 MBps)							
Disk	Seagate Barracuda	Interface	Capacity	RPM	Seek	Transfer (MBps)		Cache
						External	Internal	
	Fast-Wide (ST15150W)	SCSI-2 FastWide	4.3 GB	7200	4.2ms	20 MBps	5.9 – 8.8	1 MB
	Ultra-Wide (ST34371W)	SCSI-2 UltraWide	4.3 GB	7200	4.2ms	40 MBps	10 - 15	512 KB
Software	Microsoft Windows NT Server 4.0 SP3, NT file system and NT's *ftdisk* for striping experiments							

Table 1 Basic Hardware and Software Configuration – This system is representative of a small server or high-end desktop system at the time these studies were performed in mid-1997.

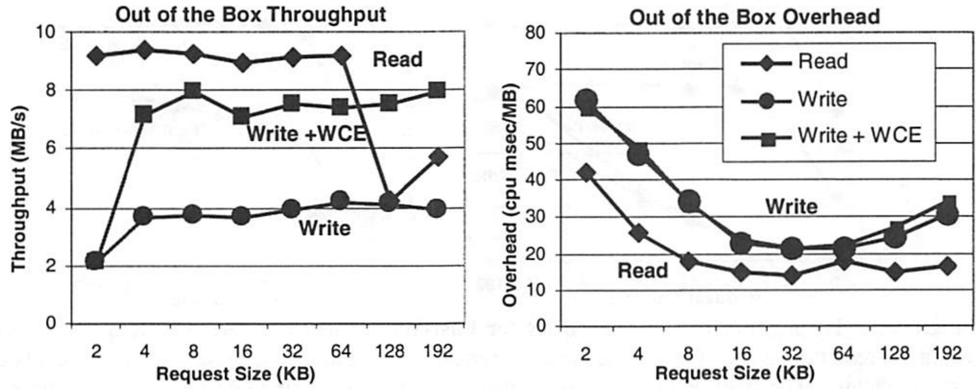


Figure 2 – Out-of-the-box Performance of a Single Ultra-Wide Drive – File system prefetching allows reads to reach full media bandwidth at small requests, although there is a sharp drop at very large request sizes. Using Write-Cache-Enable (WCE) nearly doubles write throughput. Processor cost per megabyte transferred shows that writes are more expensive than reads and overhead is minimal for requests in the 16 KB to 64 KB range.

as shown in Figure 2 – WCE approximately doubles buffered sequential write throughput. When combined with file system write buffering, this allows small requests to attain throughput comparable to large request sizes and close to the performance of reads.[1]

Small requests involve many more system calls and protection domain crossings per megabyte of data moved. With 2 KB requests, the 200 MHz processor saturates when reading writing 16 MBps. With 64 KB requests, the same processor can generate about 50 MBps of buffered file IO – exceeding the Ultra-Wide SCSI PAP. As an upper bound, this processor and memory system can generate up to 480 MBps of unbuffered disk traffic.

Write requests of 2 KB present a particularly heavy load on the system. In this case, the filesystem must read the file prior to the write-back in units of 4 KB which more than doubles the load on the system. This pre-read can be avoided by (1) issuing write requests that are at least 4 KB, or (2) truncating the file at open by specifying `TRUNCATE_EXISTING` rather than `OPEN_EXISTING` as a parameter to `CreateFile()`. When we truncated the test file on open, throughput of 2 KB writes was about 3.7 MBps, just less than that of 4 KB and larger. `TRUNCATE_EXISTING` should only be used with small, buffered requests. With 4 KB and larger requests, extending the file after truncation incurs overheads which lower throughput up to 20%. This effect is discussed further in Section 5.3.

System behavior under large reads and writes is very different. During the read tests, processor load is fairly uniform. The file system prefetches data into the cache and then copies the data to the application's buffer. The file cache buffer can be reused as soon as the data is copied to the application. During the write tests, the processor load goes through three phases. In the first phase, the application writes at memory speed, saturating the processor as it fills all available file system buffers. During the second phase, the file system must free buffers by initiating SCSI transfers. New application writes are admitted as buffers become available. The processor is about 30% busy during this phase. At the end of this phase, the application closes the file and forces the file system to synchronously flush all remaining writes - one SCSI transfer at any time. During this third phase, the processor load is negligible.

Not all processing overhead is charged to the benchmark process in Figure 2. Despite some uncertainty in the measurements, the basic trend remains: moving data with many small requests costs significantly more than moving the same data with fewer larger requests. We will return to the cost question in more detail in the next section.

3 Improving Performance - Bypassing the File System Cache

Our next experiments bypass file system buffering to more closely examine the underlying hardware performance. This section provides data on both Fast-Wide (20 MBps) and Ultra-Wide (40 MBps) disks. The Ultra-Wide disk is the current generation of the Seagate Barracuda 4LP product line and the Fast-Wide disk is the previous generation. Figure 3 shows that the devices are capable of 30% of the PAP speeds. The input file is opened with `CreateFile(,… FILE_FLAG_NO_BUFFERING|FILE_FLAG_SEQUENTIAL_SC`

[1] Enabling WCE improves performance but risks corruption if the disk fails while uncommitted data is in its cache. The on-disk cache may also be lost by SCSI bus resets [SCSI93].

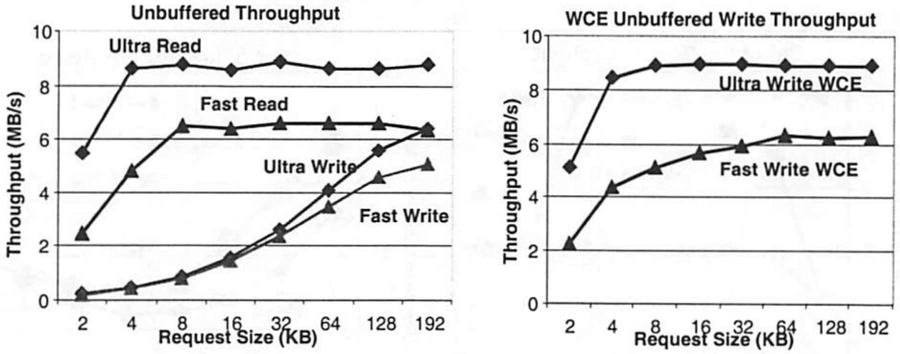


Figure 3 – Single Disk Throughput of Unbuffered IO for Fast-Wide and Ultra Drives – Requests of 8 KB and larger achieve the maximum read throughput. Write throughput is dramatically worse and increases only gradually because writes do not benefit from prefetching. The chart on the right shows that if drive write caching (WCE) is enabled, write throughput becomes comparable to read throughput. The newer Ultra drive has over a 100% advantage for small transfers, and a 50% advantage for large transfers due to its higher media transfer rate.

AN,...) and the file system performs no prefetching, no caching, no coalescing, and no extra copies. The data moves directly into the application from the SCSI adapter using direct memory access.

On large (64 KB) requests, bypassing the file system copy cuts the processor overhead by a factor of ten, from 2 instructions per byte to 0.2 instructions per byte. Unbuffered sequential reads reach the media limit for all requests larger than 8 KB. The older Fast-Wide disk requires read requests of 8 KB to reach its maximum efficiency of about 6.5 MBps. The newer Ultra-Wide drive plateaus at 8.5 MBps with 4 KB requests. Prefetching by the controller gives pipeline parallelism and allows drives to read at their media limits. Very large requests continue to perform at media rates, in contrast to the problems seen in Figure 2 with large buffered transfers.

Writes are significantly slower. The left chart of Figure 3 shows that throughput increases only gradually with request size. We observed no plateau in write throughput even for requests as large as 1 MB. The storage subsystem is completely synchronous – first it writes to the device cache, then to disk – so device overhead and latency dominate. Application requests above 64 KB are broken into multiple 64 KB requests in the IO subsystem, but these can be simultaneously outstanding at the device. The half-power write rate is achieved with a request size of 128 KB.

The right graph of Figure 3 shows that WCE compensates for the lack of file system coalescing. The WCE sequential write rates look similar to the read rates and the media limit is reached at about 8 KB for the newer disk and 64 KB for the older one. The media transfer time and rotational latency costs are hidden by the pipeline parallelism in the drive. WCE also allows the drive to perform fewer and larger media writes, reducing the total rotational latency.

Figure 4 shows the processor overhead corresponding to unbuffered sequential writes. In all cases, overheads decrease with request sizes. Requests less than 64 KB

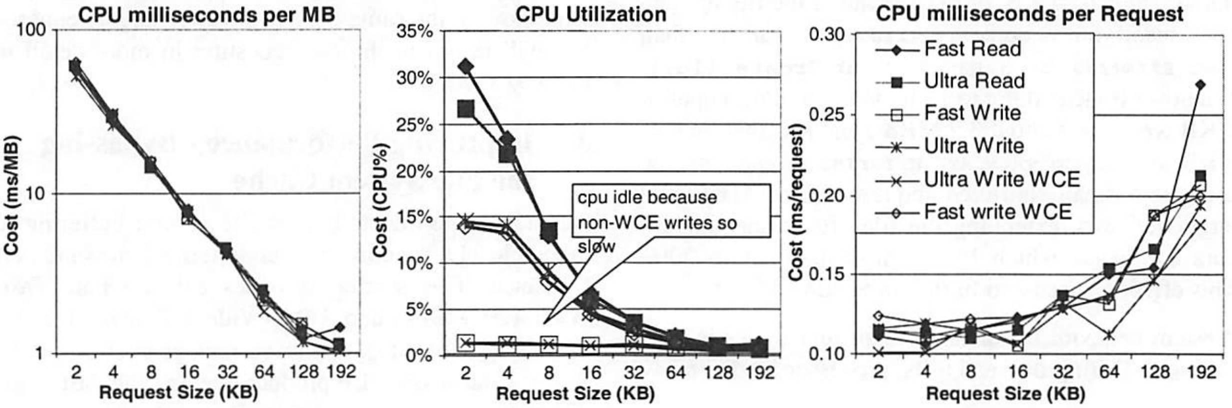


Figure 4 – Processing Cost of Unbuffered Sequential IO – The larger the request size, the more the cost of the request can be amortized. Requests of 64 KB reduce the load to less than 5%. Three drives running independent sequential streams of 2 KB requests would consume 96% of a 200 MHz Pentium Pro system.

cost about 120µs and as requests become larger, the file system must do extra work to fragment them into 64 KB requests to the device. The first chart shows the processor time to transfer each megabyte of data. Issuing many small read requests places a heavy load on the processor while larger requests amortize the fixed overhead over many more bytes. The time is similar for reads and writes regardless of the generation of the disk and disk cache setting. The center chart of Figure 4 shows the processor utilization as a function of request size. At small requests, reads place a heavier load on the processor because the read throughput is so much higher than that of writes. The processor is doing approximately the same work per byte, but the bytes are moving faster so the imposed load is higher. Finally, the chart on the right of Figure 4 shows the processor time per request. Requests up to 16 KB consume approximately the same amount of time. Since a 16 KB request moves eight times as much data as a 2 KB request, we see a corresponding 8x change. Until the request size exceeds 64 KB, larger requests consume comparable processor time. Beyond 64 KB, the processor time increases because the file system does extra work, breaking the request into multiple 64 KB transfers and dynamically allocating control structures. Note that while the cost of a single request increases with request size, the cost per megabyte always decreases.

As a rule of thumb, requests cost about 120 µs, or about 10,000 instructions. Buffered requests have an additional cost of about 2 instructions per byte while unbuffered transfers have almost no marginal cost per byte. Recall that buffered IO saturates the processor at about 50 MBps for 64 KB requests. Unbuffered IO consumes about 2.1 ms per megabyte, so unbuffered IO will saturate this system's processor at about 480 MBps. On the system discussed here, the PCI peripheral bus would have become saturated long before this point and the memory bus would be near saturation.

4 Improving Performance via Parallelism

The previous sections examined the performance of synchronous requests to a single disk. Any parallelism in the system was due to caching by the file system or disk controller. This section examines two improvements: (1) using asynchronous IO to pipeline requests and (2) striping across multiple disks to allow media transfer parallelism.

Asynchronous IO increases throughput by providing the IO subsystem with more work to do at any instant. The disk and bus can overlap or pipeline the presented load and reduce idle time. As seen above, there is not much advantage to be gained by read parallelism on a single disk. The disk is already prefetching and additional outstanding requests create only a small additional overlap on the SCSI transfer. On the other hand, WCE parallelism dramatically improves single disk write performance.

In our asynchronous IO tests, the application issues

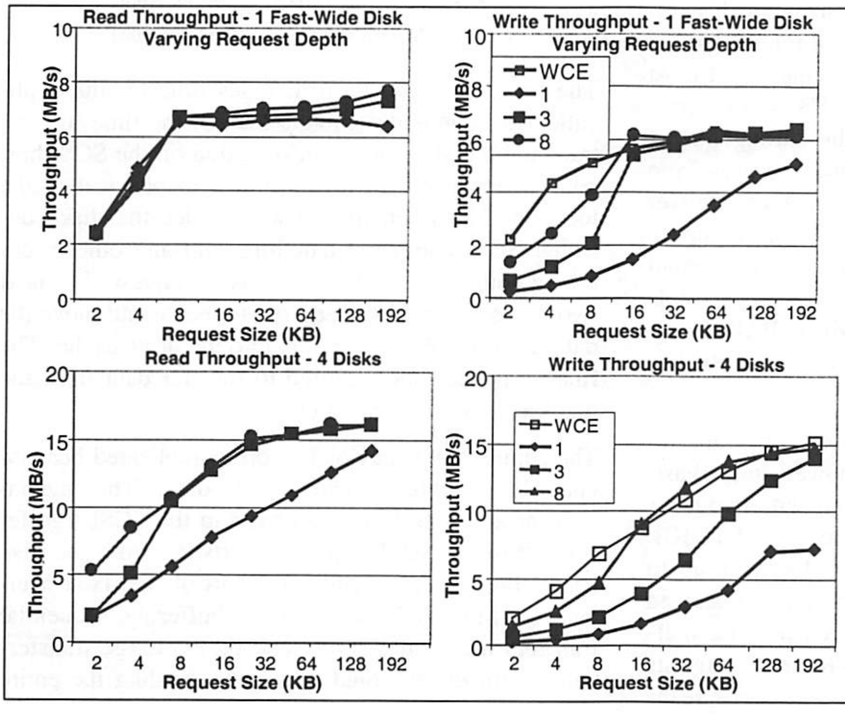


Figure 6 – Throughput of a Single Fast-Wide Drive and Four Disk Striping – The graphs show throughput for 1, 3, and 8 outstanding requests, as well as one outstanding request with WCE enabled. In the single disk case, asynchronous requests do not improve read performance because the disk controller is already prefetching. Asynchronous requests do improve write performance and, at larger request sizes, match the performance of enabling WCE. In the four drive case, asynchronous requests do improve read performance as the SCSI bus is better utilized, but writes are still substantially slower than reads.

multiple sequential IOs concurrently. When one request completes, the application asynchronously issues another as part of the IO completion routine from the earlier request, attempting to keep *n* requests active at all times. The top of Figure 6 shows the read and write throughput of a single disk as the number of outstanding requests grows from 1 to 8. Read throughput is not much changed, while write throughput improves dramatically. Reads reach the half-power point with 4 KB requests. Writes need 3-deep 16 KB requests or 8-deep 8 KB requests to reach the half-power point, which represents a 4x improvement over synchronous 8 KB writes. For requests of 16 KB and more, 3-deep writes are comparable to the throughput of when WCE.

As more disks are added to the system, asynchronous IO gives significant benefits for reads and large transfers as well as smaller writes. The lower charts of Figure 6 show the results when the file is striped across four Fast-Wide SCSI disks on a single host bus adapter (and single SCSI bus). *ftdisk* is used to bind the drives into a stripe set and each successive disk gets the next 64 KB file chunk in round-robin fashion. At 4 KB and 8 KB requests, increasing request depth increases throughput as requests are spread across multiple disks. With a chunk size of 64 KB, 8-deep 8 KB requests will have IOs outstanding to more than one drive 7/8 of the time, approximately doubling the throughput. Smaller request depths distribute the load less effectively – with only two requests outstanding, IOs are outstanding to more than one drive only 1/4 of the time. Similarly, smaller request sizes are less effective since more requests are required for each stripe chunk. At 4 KB requests and 8 deep requests, at most two drives are used, and this only 3/8 of the time. Striping large requests improves the throughput of both reads and writes and the bottleneck moves from the disk media to the SCSI bus. Each disk can deliver about 6 MBps, so four disks should deliver up to 24 MBps. The experiments all saturated at about 16 MBps, so the RAP bandwidth of our Fast-Wide SCSI subsystem is 80% of the 20 MBps PAP. Ultra-Wide SCSI (not shown) also delivers 75% of PAP or about 30 MBps.

Both large request sizes and multiple disks are required to reach the SCSI bus half-power point. Fast-Wide SCSI can reach half-power points with two disks at read requests of 8 KB and write requests of 16 KB. Using 64 KB or larger requests, transfer rates up to 75% of the advertised bus bandwidth can be observed with three disks. Ultra-Wide SCSI reaches the half-power point with three disks and 16 KB read requests or 64 KB write requests. Only with very large reads can we reach 75% of the advertised bandwidth. The bus protocol overheads and actual data transfer rates do not scale with advertised bus speed. Further experiments show that three Ultra-Wide disks saturate a single Ultra-Wide SCSI bus. Two buses support a total of six disks and a maximum read throughput of 60 MBps. When a third adapter and three more disks are added, the PCI bus limit is reached and the configuration achieves a total of only 72 MBps – just over the half-power point of the PCI. Adding a fourth adapter shows no additional improvement, although the combined SCSI bandwidth of 120 MBps would seem to be well within the advertised 133 MBps of the PCI. While the practical limit is likely to depend on the exact hardware, the PCI half-power point appears to be a good goal.

5 Detailed Performance Measurements

The previous sections provided an overview of a typical storage system and discussed a number of parameters affecting sequential I/O throughput. This section investigates the hardware components in order to explain the observed behavior.

5.1 Disk Controller Caching and Prefetching

A simple model for the cost of a single disk read assumes no pipelining and separates the contributing factors:

$$\begin{aligned} \mathit{Request\_Service\_Time} = \mathit{Fixed\_Service\_Time} \\ + \mathit{Disk\_Seek\_Time} \\ + (\mathit{Transfer\_Size}/\mathit{Media\_Transfer\_Rate}) \\ + (\mathit{Request\_Size}/\mathit{SCSI\_TransferRate}) \end{aligned}$$

The fixed overhead term includes time for the application to issue and complete the IO, the time to arbitrate and transfer control information on the SCSI bus, converting the target logical block to physical media location. The fixed time also includes the disk controller SCSI command handling, and any other processing common to any data transfer request. The next two terms are the time required to locate and move the data from the physical media into the drive cache. The final term the time required to transfer data from the disk cache over the SCSI bus.

The actual disk behavior is more complicated because controllers prefetch and cache data. The media-transfer and seek times can overlap the SCSI transfer time. When a SCSI request is satisfied from the disk cache, the seek time and some part of the fixed overhead is eliminated. Even without buffering, sequential transfers incur only short seek times. Large transfers can minimize rotational latency by reading the entire track – full-track transfers can start with the next sector to come under the read-write head.

At the extremes, some simplifications should occur. For small (2KB) requests, the fixed overhead dominates the transfer times (> 0.5 ms). For large (> 32 KB) requests, the media-transfer time (> 8 ms) dominates. The fixed overhead is amortized over a larger number of bytes and the SCSI transfer rate is faster (> 2x) than the media-transfer rate. We measured the fixed overhead component for three generations of Seagate drives: the Narrow 15150N, the Fast-Wide 15150W, and the Ultra-Wide 34371W. Table 3 shows the results. The cache hit data were obtained by reading the same disk blocks repeatedly. The prefetch hit column was obtained using the benchmark program to sequentially read a 100 MB file. To ensure that the prefetched data would be in the drive cache at all times, a delay was inserted between SCSI requests for those transfers marked with asterisks (*).

Table 3 – Variation across disk generation - The elapsed time in ms for a cache hit and prefetch hit of varying request sizes directly. Times are measured from an ASPI driver program issuing SCSI commands and bypassing the NT file system. For the large request sizes, the drive is given sufficient time between requests to ensure that the request is always satisfied from prefetch buffers and not limited by media transfer rates. Surprisingly, the cache hit times are always larger than the prefetch hit times.

	Narrow-ST15150N		Fast-Wide-ST15150W		Ultra-Wide-ST34371W	
Size	*Cache Hit*	*Prefetch Hit*	*Cache Hit*	*Prefetch Hit*	*Cache Hit*	*Prefetch Hit*
5K	0.96	0.56	0.93	0.59	8.14	0.30
1K	1.01	0.63	0.97	0.59	8.14	0.32
2K	1.11	0.75	1.02	0.58	8.14	0.34
4K	1.33	0.93	1.13	0.61	8.13	0.40
8K	1.75	1.38	1.36	0.86	8.13	0.51
16K	2.63	2.25	1.81	1.31*	8.13	0.74*
32K	4.35	3.93*	2.75	2.25*	8.13	1.22*
64K	16.50	7.30*	16.50	4.05*	8.15	2.15*

We expected that the cache hit case would be a simple way to measure fixed overhead. The data are already in the drive cache so no media operation is necessary. The results, however, tell a different story. The prefetch hit times are uniformly smaller than the cache hit times. The firmware appears to be optimized for prefetching – it takes longer to recognize the reread as a cache hit. In fact, the constant high cache hit times of the 34371W imply that this drive does not recognize the reread as a cache hit and rereads the same full track at each request. At 64 KB, the request spans tracks; the jump in the 15150 drive times may also be due to media rereads.

The prefetch hit data follow a simple fixed cost plus SCSI transfer model up through 8 KB request sizes. The SCSI transfer time was computed using the advertised bus rate. The 15150 drives (both Narrow and Fast-Wide) have fixed overhead of about 0.58 milliseconds; the 34371W drive (Ultra-Wide) has overhead of about 0.3 milliseconds.

At larger requests, no simple model applies. At 64KB, the computed SCSI transfer times do not account for the full prefetch hit time and the remainder is greater than the observed fixed overhead times. The media-transfer rate is not the limit because of the delay between requests. Without the delay, the measurements showed larger variation and the total time was not fully accountable to media transfer. The total time appears to be due to a combination of prefetch hit and new prefetch. A 64KB request may span up to three disk tracks and at least that many prefetch buffers. Whether or not the disk prefetches beyond the track necessary to satisfy the current request is unclear and likely to be implementation specific. Whether or not the disk can respond promptly to a new SCSI request when queuing a new prefetch is also unclear.

Intelligence and caching in the drive allows overlap and parallelism across requests so simple behavioral models no longer capture the behavior. Moreover, drive behavior changes significantly across implementations [Worthington95]. While the media-transfer limit remains a valid half-power point target for bulk file transfers, understanding smaller scale or smaller data set disk behavior seems difficult at best.

5.2 SCSI Bus Activity

We used a bus analyzer to measure SCSI bus activity. Table 4 summarizes the contribution of each protocol cycle type to the total bus utilization while reading the standard 100 MB file. Comparing the first two columns, small requests suffer from two disadvantages:

Small requests spend a lot of time on overhead. Half the bus utilization (30% of 60%) goes to setting up the transfer. There are eight individual 8KB requests for each 64KB request. This causes the increased arbitration, message, command and select phase times.

Table 4 – SCSI Activity by Phase - For 8KB requests, only 45% of the SCSI bus is data transfer (column 2). The balance goes to SELECT/RESELECT activity and parameter messaging. Larger requests make much more efficient use of the bus - for 64KB requests, utilization drops by half and data transfer makes up almost 90% of that time (column 3). When more disks are added, this efficiency drops somewhat in favor of more message traffic and SELECT activity. The three-disk system reaches over 99% bus utilization and consumes significantly more time in SELECT (column 5).

Phase	8KB Requests	64KB Requests		
	1 Disk	1 Disk	2 Disks	3 Disks
Arbitrate	1.1%	0.4%	0.6%	0.4%
Arbitrate Win	0.6%	0.2%	0.3%	0.2%
Reselect	0.2%	0.1%	0.1%	0.1%
Select	25.2%	0.2%	0.8%	4.4%
(Re)Select End	0.3%	0.1%	0.1%	0.1%
Message In	18.5%	7.4%	11.4%	9.1%
Message Out	5.5%	1.4%	2.8%	3.6%
Command	2.1%	0.5%	1.0%	1.1%
Data In	44.9%	89.3%	82.2%	80.4%
Data In End	0.7%	0.3%	0.4%	0.2%
Data Out	-	-	-	-
Data Out End	-	-	-	-
Status	0.7%	0.2%	0.3%	0.4%
Bus Utilization	**59.8%**	**30.1%**	**67.8%**	**99.3%**

Small requests spend little time transferring user data. At 64KB, 90% of the bus utilization is due to application data transfer. At 8KB, only 45% of the bus time is spent transferring application data.

The last two columns of Table 4 show the effects of SCSI bus contention. Adding a second disk doubles throughput but bus utilization increases 125%. The extra 25% is spent on increased handshaking (SELECT activity and parameter passing). The SCSI adapter is pending requests to the drives and must reSELECT the drive when the request can be satisfied by the drive. More of the bus is consumed coordinating communication among the disks. Adding a third disk increases throughput and fully consumes the SCSI bus, as discussed in Section 3. The SELECT activity increases again, further reducing the time available for data transfer. The overall bus efficiency decreases as disks are added because more bus cycles are required coordinate among the drives.

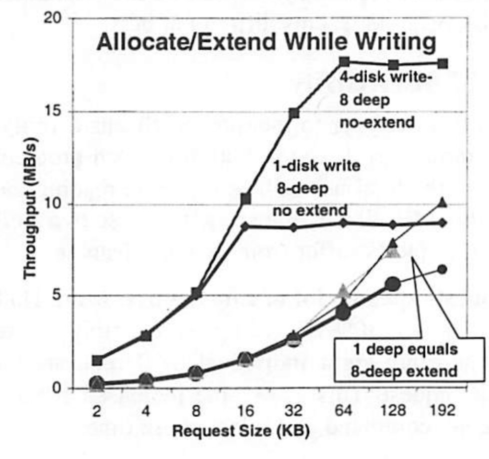


Figure 12 – File Allocate/Extend Behavior – When a file is being extended (new space allocated at the end), NT forces synchronous write behavior to prevent requests from arriving at the disk out-of-order. Security mandates that the value zero be returned to a reader of any byte which is allocated but has not yet been written. The file system must balance performance against the need to prevent programs from allocating files and then reading data from files deallocated by other users. The extra allocate writes dramatically slow write performance.

5.3 Allocate

Unbuffered file writes have a serious performance pitfall. The NT file system forces unbuffered writes to be synchronous whenever a file is newly created and whenever the file is being extended either explicitly or by writing beyond the end of file. This synchronous write behavior also happens for files that are truncated (specifying the `TRUNCATE_EXISTING` attribute at `CreateFile()` or after open with `SetEndOfFile()`).

As illustrated in Figure 12, allocation severely impacts asynchronous IO performance. The file system allows only one request outstanding to the volume. If the access pattern is not sequential, the file system may actually zero any new blocks between requests in the extended region. Buffered sequential writes are not as severely affected, but still benefit from preallocation. Extending a file incurs at most about a 20% throughput penalty with small file system buffered writes. There is one notable exception. If you use tiny 2 KB requests, allowing the file system to allocate storage dynamically actually improves performance. The file system does not pre-read the data prior to attempting to coalesce writes.

To maximize asynchronous write performance, you should preallocate the file storage. If the space is not pre-allocated, the NT file system will first zero it before letting your program read it.

5.4 Alignment

The NT 4.0 file system (using the *ftdisk* mechanism) supports host-based software RAID 0, 1, and 5. A fixed stripe *chunk size* of 64 KB is used to build RAID0 stripe sets. Each successive disk gets the next 64KB chunk in round-robin fashion. The chunk size is not user-settable and is independent of the number or size of the stripe set components. The file system allocates file blocks in multiples of the file system *allocation unit* chosen when the volume is formatted. The allocation unit defaults to a value in the range of 512 bytes to 4 KB depending on the volume size. The stripe chunk and file system allocation units are totally independent; NT does not take the chunk size into account when allocating file blocks. Thus, files on a multiple-disk stripe set will almost always be misaligned with respect to stripe chunk size.

Figure 13 shows the effect of this misalignment. Alignment with the stripe chunk improves performance by 15-20% at 64 KB requests. A misaligned 64 KB application request causes two disk requests (one of 12 KB and another of 52 KB) that must both be serviced before the application request can complete. As shown earlier, splitting application requests into smaller units reduces drive efficiency. The drive array and host-bus adapter sees twice the number of requests and some of those requests are small. As the SCSI bus becomes loaded, the performance degradation becomes more noticeable. When requests are issued 8-deep, there are eight 64 KB requests active at any given time. In the misaligned case, there are 16 requests of mixed 12 KB and 52 KB sizes to be coordinated.

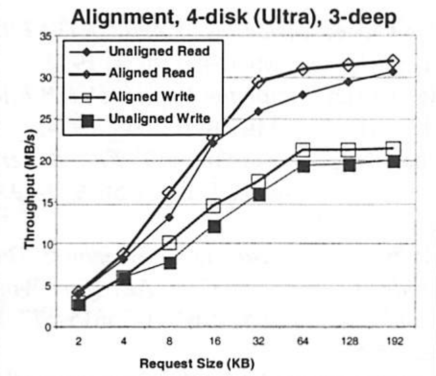


Figure 13 – Alignment Across Disks in a Stripe Set – The performance of a file aligned to the stripe chunk is compared to a file that is mis-aligned by 12KB. If requests split across stripe set step boundaries, read performance can be reduced by nearly 20% and writes by 15%. The effect is more pronounced with 8 requests outstanding because there is more activity on the SCSI bus and more contention.

Misalignment can be avoided by using the NT file system **`format`** command at the command prompt rather than the Disk Administrator application.[2] Disk Administrator limits the allocation size to 512, 1024, 2048, or 4096 bytes, while format command allows increments up to 64 KB. The cost of using a 64 KB allocation unit is the potential wasted disk space if the volume contains many small files; the file system always rounds the file size to the allocation unit.

6 Summary and Conclusions

The NT 4.0 file system out-of-the-box sequential IO performance is good: reads are close to the media limit and writes are near the half-power point. This performance comes at some cost; the file system is copying every byte, and coalescing disk requests into 64 KB units. Write throughput can be nearly doubled by enabling WCE, although this risks data corruption, and similar results can be achieved by using large requests and issuing asynchronous requests. NT file striping across multiple disks is an excellent way to increase throughput, but in order to take advantage of the available parallelism; striping must be combined with large and deep asynchronous requests.

An application can saturate a SCSI bus with three drives. By using multiple SCSI busses, it can saturate a PCI bus. By using multiple PCI buses, it could saturate the processor bus and memory subsystem. If the system configuration is balanced (disks do not saturate busses, busses do not saturate), the NT file system can reach the half-power point. In fact, applications can reach the sum of the device media limits by using a combination of (1) large request sizes, (2) deep asynchronous requests, (3) WCE, (4) striping, and (5) unbuffered IO.

Write performance is often significantly lower than read performance. The main pitfalls in writing files are: (1) if a file is not already allocated, the file system will force sequential writing in order to prevent applications from reading data left on disk by the previous file using that disk space (2) if a file is allocated but not truncated on open, then smaller than 4 KB buffered writes will first read a 4 KB unit and then overwrite (3) if the stripe chunk size is not aligned with the file system allocation size, large requests are broken into two smaller requests split across two drives, which doubles the number of requests to the drive array.

[2] A command of the form '**`format e: /fs:ntfs /a:64k`** ' to create a file system with 64 KB allocation.

The measurements suggest a number of ways of doing efficient sequential file access:

- Larger requests are faster. Requests should be at least 8 KB, 64 KB if possible.

- Small requests consume significantly more processor time per byte than larger ones. 2 KB requests consume more than 30% of the processor while 64 KB requests both go faster and consume only 3% of the processor.

- If an application absolutely must make small requests, double buffering is not enough parallelism. There are noticeable gains through 8-deep requests.

- Write-Cache-Enable at drives provides significant benefits for small requests. Issuing three-deep asynchronous requests comes close to WCE performance for larger requests.

- Three disks can saturate a SCSI bus, whether Fast-Wide (15 MBps max) or Ultra-Wide (31 MBps max). Adding more disks than this to a single bus does not improve performance.

- File system buffering coalesces small requests into 64 KB disk requests for both reads and writes. This provides significant performance improvement for requests smaller than 64 KB.

- At 64 KB and larger requests, file system buffering degrades performance from the non-buffered case.

- When possible, files should be preallocated to their eventual maximum size.

- Extending a file while writing forces synchronization of the requests and significantly degrades performance.

This paper provided a basic tour of the parameters that affect sequential IO performance in NT and examined the hardware limitations at each stage in the IO pipeline. We have provided guidance on how applications can take advantage of the parallelism in the system and overlap requests for best performance. We have also shown that while out-of-the-box performance is reasonable for some workloads, there are a number of parameters that can make a factor of two to ten difference in overall throughput.

Many areas are not discussed here and merit further attention. Programs using asynchronous I/O have several options for managing asynchronous requests, including completion routines, events, completion ports, and multi-threading. Our benchmark uses completion routines in an otherwise single-threaded program and we have not explored the tradeoffs and overheads of using the other methods. This analysis focused on a single benchmark application issuing a single stream of sequential requests. A production system is likely to have several applications competing for storage resources. This complicates the model since the device array no longer sees a single sequential request stream.

7 Acknowledgements

Tom Barclay did the initial development of the **iostress** benchmark used in all these studies. Barry Nolte and Mike Parkes pointed out the importance of the allocate issue. Doug Treuting, Steve Mattos and others at Adaptec helped us understand SCSI details and the how the device drivers work. Bill Courtright, Stan Skelton, Richard Vanderbilt, Mark Regester of Symbios Logic generously loaned us an array, host adapters, and their expertise. Brad Waters, Wael Bahaa-El-Din, and Maurice Franklin shared their experience, results, tools, and hardware laboratory. They helped us understand NT performance issues and gave us feedback on our preliminary measurements. Will Dahli helped us understand NT configuration and measurement. Don Slutz and Joe Barrera gave us valuable comments, feedback and help in understanding NT internals. Finally, we thank the anonymous reviewers for their comments and our shepherd, Brad Chen, for his help on short notice.

8 References

[Custer92] Helen Custer, *Inside the WindowsNT™ File System*, ISBN 1-55615-481, Microsoft Press, 1992.

[Custer94] Helen Custer, *Inside the WindowsNT™ File System*, ISBN 1-55615-660, Microsoft Press, 1994.

[Nagar97] Rajeev Nagar, *Windows NT File System Internals: A Developer's Guide*, ISBN 1-565922-492, O'Reilly & Associates, 1997.

[Richter95] Jeffrey Richter, *Advanced Windows: The Developers Guide to the Win32™ API for WindowsNT™ 3.5 and Windows 95,* ISBN 1-55615-677-4, Microsoft Press, 1995.

[Schwaderer96] W. David Schwaderer, Andrew W. Wilson Jr. *Understanding I/O Subsystems, First Edition,* ISBN 0-9651911-0-9, Adaptec Press, Milpitas, CA, 1996.

[SCSI93] *ANSI X3T9.2 Rev 10L, 7-SEP-93*, American National Standards Institute, *www.ansi.org*.

[Seagate97] Seagate Technology "Barracuda Family Product Specification" *www.seagate.com.*

[Worthington95] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes, "On-Line Extraction of SCSI Disk Drive Parameters", *Proceedings of ACM Sigmetrics*, May 1995, pp. 146-156.

Scalability of the Microsoft Cluster Service

Werner Vogels, Dan Dumitriu, Ashutosh Agrawal, Teck Chia, Katherine Guo
Department of Computer Science, Cornell University[†]
vogels@cs.cornell.edu

Abstract

An important argument for the introduction of software managed clusters is that of scale: By constructing the cluster out of commodity compute elements, one can, by simply adding new elements, improve the reliability of the overall system in terms of performance and in availability. The limits to how far such a cluster can be scaled seems to be dependent on the scalability of its management software, which in its core has a collection of distributed algorithms to guarantee the correct operation of the cluster. The complexity of these algorithms makes them a vulnerable component of the system in terms of their impact on the overall scalability of the system.

This paper examines two of the distributed components of the Microsoft Cluster Service [8] that are most likely to have an impact on its scalability: the membership and the global update managers. The first sections of the paper will provide some general background on these distributed services and scalability issues. After that the algorithms used to implement these service are described in detail and an analysis of their impact on scalability is given. The scalability analysis is based on an off-line analysis of the algorithms as well as the results of on-line experiments on a cluster with a, in MSCS terms, large number of nodes.

1 Distributed Management

In distributed management software two components are considered basic building blocks: a consistent view about which nodes are on-line, and the ability to communicate with these nodes in an all-or-nothing fashion [2].

The first building block is captured in a *membership* service: all nodes participate in a consensus algorithm to agree on the current set of nodes that are up and running. The system makes use of a failure detection mechanism that monitors heartbeat signals or actively polls other nodes in the system. The failure detector will signal the membership service whenever it suspects the failure of a node in the system. The membership service will react to this by triggering the execution of a distributed algorithm at all the nodes in the system, in which they agree upon which nodes have failed and which are still available. The joining of a new member in the system, does not require the nodes to run the agreement protocol, but can often be handled through a simple update mechanism.

The second fundamental component provides a special communication facility, with guarantees that exceed the properties provided by regular communication systems. Often in the process of managing a distributed system it is necessary to provide the same information to a set of nodes in the system. We can simplify the software design of many of the components on the receiving side of this information if we can guarantee that if one node receives this information, that all nodes will receive it. This *atomicity* guarantee allows nodes to act immediately upon reception of a message, without the need for additional synchronization. Often this atomicity guarantee is not sufficient for a system, as it does not only need be assured that all nodes will receive the update, but that all nodes will see the updates in the same order. This *total order* property makes the communication module a very powerful mechanism in the control of the distributed operation of the distributed system.

2 Practical Scalability

This paper examines the membership and communication services of the Microsoft Cluster Service (MSCS) with an emphasis on their impact on the scalability of the system. MSCS, as shipped, officially supports 2 nodes, but in reality the software can be run on a 16-node NT server cluster. At Cornell the software is extended to run on 32 nodes and a research project is underway to make the system scale to larger numbers.

Making systems scale in practice centers around the use of mechanisms to reduce the dependency of the algorithms on the number of nodes. In the past two approaches have been successful in finding solutions to problems of scale: The first is to reduce the

[†] The reliable cluster computing research of the Reliable Distributed Computing Group at the department of Computer Science at Cornell University is supported by DARPA/ONR under contract N0014-96-1-10014 and by Intel Corporation and Microsoft Corporation.

synchronous behavior of the system by designing messaging systems and protocols that allow high levels of concurrent operation, for example by de-coupling the sending of messages from the collecting of acknowledgements. The second approach is to reduce the overall complexity of the system. By building the system out of smaller (semi-)autonomous units and connecting these units through hierarchical methods, growing the overall system has no impact on the mechanism and protocols used to make the smaller units function correctly.

A third, more radical approach, which is under development at Cornell, makes use of gossip based dissemination algorithms. These techniques significantly reduce the number of messages and the amount of processing needed to reach a similar level of information sharing among the cluster nodes.

Given that cluster systems such as MSCS are used for enterprise computing, any instability of the system can have severe economic results. There is a continuous tradeoff between responsive failure handling and the cost of an erroneous suspicion. The system needs to detect and respond to failures in a very timely matter, but designers may choose a more conservative approach given the significant cost of an unnecessary reconfiguration of the system, caused by an incorrect failure suspicion. In general cluster server systems run compute and memory intensive enterprise applications and these systems experience a significant load at times, reducing the overall responsiveness. Scaling failure detection needs intelligent mechanisms for fault investigation [6,11] and requires the failure detectors to be able to learn and adapt to changes [7].

3 Scalability goals of MSCS

The Microsoft Cluster Service is designed to support two nodes, with a potential to scale to more nodes, but in a very limited way. MSCS successfully addresses the needs of these smaller clusters. The cluster management tools are a significantly improvement over the current practice and they are a major contribution to the usability of clusters overall.

The research reported here is concerned with scaling MSCS to larger numbers of nodes (16 - 64, or higher), which is outside of the scope of the initial MSCS design. There are three areas of interest:

1. Can the currently used distributed algorithms be a solid foundation for scalable clusters?
2. Are there any architectural bottlenecks that should be addressed if MSCS needs to be scalable?
3. If MSCS is extended with development support for cluster aware applications are the current distributed services a good basis for these tools?

This paper should not be seen as criticism of the current MSCS design. Within the goals set for MSCS it functions correctly and will scale to numbers larger then originally targeted by the cluster design team.

4 Cluster Management

The algorithms used in MSCS for membership and total ordered messaging are a direct derivative of those developed in the early eighties for Tandem as used in the NonStop systems [3,4]. Nodes in a Tandem system communicated via pairs of proprietary inter-processor busses, which, in 1985, provided a 100 Mbit/second transfer rate. Parts of the messaging side of the algorithms was implemented in interrupt handlers to provide minimal system overhead.

Although MSCS has a kernel module that implements some of the messaging and failure detection, the membership and global update algorithms are implemented in an NT service, the *Cluster Service*, which runs at user level. The Cluster Service holds in total 11 managers, each responsible for a different part of the cluster service functionality. Next to the membership and communication managers, there are managers for resource and failover management, for logging and checkpointing, and for configuration and network management.

In the following sections three of the components are examined in detail: first the kernel module which holds the cluster communication and failure detection functionality. Secondly the join process and the failure reconfiguration of the membership module are analyzed. The last analysis is that of the global update communication module.

5 Cluster Network

MSCS provides a kernel based cluster network interface, *ClusNet*, which presents a uniform interface to networks available for intra-cluster communication. ClusNet supports basic datagram communication to each of the nodes, using an addressing scheme based on simple node identifiers which are assigned to nodes when they are first configured for use in the cluster. To support reliable communication ClusNet provides a transport interface used by MS-RPC.

ClusNet is capable of managing a redundant networking infrastructure, automatically adapting packet routing in case of network failure.

5.1 Node Failure Detection

MSCS implements its Failure Detection (FD) mechanism using heartbeats. Periodically every node sends a sequenced message to every other node in the cluster, over the networks that are marked for internal communication. Whenever a node detects a number of consecutive missing heartbeats from another node it sends an event to the cluster service which uses this event to activate the membership reconfiguration module.

In the current MSCS configuration heartbeats are sent every 1.2 seconds and the detection period for a node suspicion is 7.2 seconds (6 missed heartbeats). The timing values are not adaptive.

The cluster network module does not exploit any broadcast or multicast functionality, and thus each heartbeat results in (*number_of_nodes-1*) point-to-point datagrams. In our test setup of 32 nodes, the cluster background traffic related to heartbeats is 800 messages per second. With 32 nodes active and an otherwise idle network the mechanism works flawless and the packet loss observed was minimal. Tests which replaced the Fast-Ethernet switches with hubs showed that the packet trains sometimes caused significant Ethernet-level collisions on the shared medium. Adding processing load to the systems resulted in variations in the inter-transmission periods. False suspicions were never seen.

When adding processing load and additional load on the network frequently single heartbeat misses where observed, but the values for generating a failure suspicion event so conservative that never any false suspicions were generated.

6 Node Membership

The MSCS membership manager is designed into two separate functional modules: the first handles the joining of nodes and a second, *regroup*, implements the consensus algorithm that runs in case of a node failure.

6.1 Join

The join algorithm starts with a discovery phase in which the joining node attempts to find other nodes in the cluster. If this fails the node tries to form a cluster by itself, the details of the *cluster-form* operation are outside of the scope of the paper. After the node has discovered which cluster nodes are currently running it selects one of the nodes and petitions for membership of the cluster. The selected node, dubbed the *sponsor*, announces the new node to all active cluster members, transfers all the up-to-date cluster configuration to the new node, and waits for the node to become active. The different phases of the join and their distributed complexity are described in detail in the following paragraphs

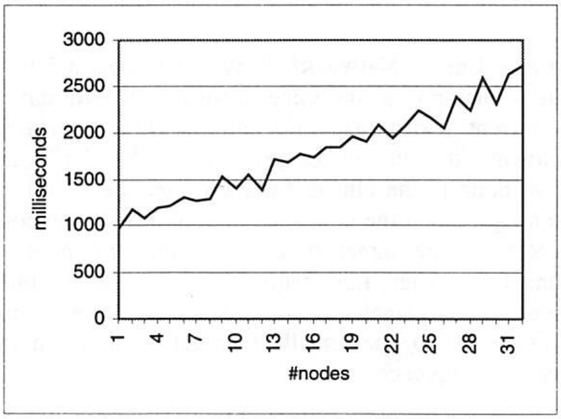


Figure 1. Join latency under ideal conditions

Phase 1: **Discovery**. When a cluster service starts is attempts to connect to each of the other known nodes in the cluster, using RPC over a regular UDP transport. This sponsor discovery mechanism has a high degree of concurrency: a thread is started for each connection probe. The joiner waits for all threads to terminate, which occurs after the RPC binding operation fails after a time-out or when a connection is established. As the joiner waits for all threads to terminate, the delay the joining node experiences is based on the time-out period of an RPC connection to a single node that is not up. The timeout value for RPC out-of-the-box is approximately 30 seconds, but it can be manipulated to reduce the discovery phase to 10 seconds.

In all observed cases, the joining node always selected the holder of the cluster IP address to sponsor its join. The cluster IP address is a single address that migrates to a node that functions as the access point for administrative purposes: if the cluster is running there is *always* a node that holds this IP address. By modifying the startup phase to start by attempting to connect to this address first before probing all the other nodes, it is possible to reduce this phase of the join process to under a second. This approach also avoids starting a number of threads that is equal to the number of nodes in the cluster.

Phase 2: **Lock.** From the nodes that are up, the joiner selects one node to *sponsor* its membership in the cluster. The first action by the sponsor is to acquire a distributed global lock to ensure that only a single join is in progress. Acquiring of the lock is performed using a global update (GUP) method.

The use of GUP makes this phase is dependent on the number of active nodes. Details on the performance and scalability of GUP can be found in section 7.

Phase 3: **Enable Network**: Using a sequence of 5 RPC calls to the sponsor the joiner retrieves all information on current nodes, networks and network interfaces. Following this the joiner performs an RPC to each active node in the cluster for each interface a node is listening on, and the contacted node in return performs an RPC to the joiner to enable symmetric network channels. After this sequence the node security contexts are established which again requires the joining node to contact all other active nodes in the cluster, in sequence.

This phase depends on the number of active nodes in the cluster. An unloaded 31 nodes cluster, on average, performs this sequence of RPC's in 2-4 seconds. On a moderately loaded cluster, frequently this phase takes longer then 60 seconds, causing the join operation to time-out at the sponsor, resulting in an abort of the join.

Phase 4: **Petition for Membership:** The joiner requests the sponsor to insert the node into the membership. This is a 5-step process directed by the sponsor.

1. The *sponsor* broadcasts to all current members the identification the joining node.
2. The sponsor sends the membership algorithm configuration data to the joiner
3. The sponsor waits for the first heartbeat from the new joiner.
4. The sponsor broadcasts to all current members that the node is alive
5. The sponsor notifies the joiner that it is inserted in the membership

The broadcasts are implemented as series of RPC calls, one to each active node in the cluster. On an unloaded cluster and network the serialized invocation of RPC to 30 nodes takes between 100 and 150 milliseconds. When loading the systems with compute and IO tasks, the RPC times vary widely from 3 millisecond to 3 second per RPC. Broadcast rounds to all 30 nodes were observed taking more then 20 seconds to complete (with exceptions up to 1 minute). As this phase is under control of the sponsor the join is not aborted because of a time-out. It can abort on a communication failure with any of the nodes.

In step 3 the detection of the new heartbeat is delegated to ClusNet, which performs checks every 600 millisecond, resulting in an average waiting period between 0.6 and 1.2 seconds

Phase 5: **Database synchronization.** The joiner synchronizes its configuration database with the sponsor. In the experimental setup this database was of minimal size and never out-of-date. As the retrieving of the database updates is not depended on cluster size, not further tests were performed in this phase.

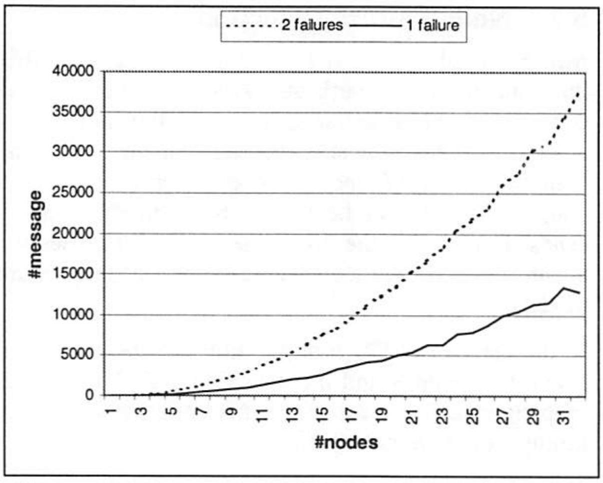


Figure 2. Number of messages in the system during regroup

Phase 6: **Unlock**. The newly joined node uses its access to the global update mechanism to broadcast to all nodes that it now is full operation and that the global lock should be released.

The join operation is very much dependent on the number of nodes in the system. Figure 1 show the times for a join under optimal conditions. All RPC calls in the algorithms are serialized and at minimum there are *(10 + 7 * number_of_nodes)* calls. Joining the 32nd node to the cluster requires at least 227 sequential RPC's. This approach collapses under load, frequently it is impossible to join any nodes if only a moderate load is placed on the nodes and the system has more then 10-12 nodes.

6.2 Regroup

Upon the receipt of a node failure event generated by ClusNet the Cluster Service starts the reconfiguration algorithm, dubbed *regroup*. The algorithm runs in 5 phases, with the transition to each new phase determined after its is believed that all other nodes have finished this phase, or when, in the first two phases, timers expire.

During regroup the nodes periodically (300ms) broadcast their current state information to all other nodes using unreliable datagrams. The state is a collection of bitmasks, one for each phase, describing whether a node has indicated it has passed a phase. It is not necessary for each node to have heard for each other node in a phase; information about which other nodes a certain node has heard of is shared. For example if node 1 indicates that it has received a regroup message from node 2, node 3 uses this without that it actually needs to receive a message from node 2 in that phase. Also included in the state is a connectivity matrix in which nodes record whether they have seen messages from the other nodes and what connectivity information has been recorded by the other nodes.

The 5 phases of the regroup algorithm are the following:

Phase 1: **Activate.** Each node waits for a local clock tick to occur so that it knows that its timeout system can be trusted. After that it starts sending and collecting status messages. It advances to the next stage if

1. All current members have been detected to be active (e.g. there was a false suspicion),
2. If there is one single failure and a minimal time-out has passed or,
3. When the maximum waiting time has elapsed and several members have not yet responded.

The minimum timeout for phase 1 is 2.4 second, if all but one node have responded in this time period it is assumed that there was a single failure and the algorithm moves to the next phase. If multiple nodes do not respond, the algorithm waits for 9.6 seconds to move to the next phase. If for some reason the regroup algorithm times out in a different phase or when there are cascading starts of the regroup algorithm at several nodes, the algorithm executes in *cautious* mode and always waits for the maximum timeout to expire.

Phase 2: **Closing.** This stage determines whether partitions exist and whether the current node is in a partition that should survive. The rules for surviving are:

1. The current membership contains more than half the original membership.
2. Or, the current membership has exactly half the original members, and there are at least two members in the current membership and this membership contains the tie breaker node that was selected when the cluster was formed.
3. Or, the original membership contained exactly two members and the new membership only has one member and this node has access to the quorum resource.

After this the new members select a tie breaker node to use in the next regroup execution. This tiebreaker then checks the connectivity information to ensure that the surviving group is fully connected. If not it prunes those members that do not have full connectivity. It records this pruning information in its regroup state, which is broadcast to all other nodes. All move to stage 3 upon receipt of this information.

In case of incomplete connectivity information the tiebreaker waits for an additional second to allow all nodes to respond.

Phase 3: **Pruning.** All nodes that have been pruned because of lack of connectivity halt in this phase. All others move forward to the first cleanup phase once they have detected that all nodes have received the pruning decision (e.g. they are in phase 3).

Phase 4: **Cleanup Phase One.** All surviving nodes install the new membership, mark the nodes that did not survive the membership change as down, and inform the cluster network to filter out messages from these nodes. Each node's Event Manger then invokes local callback handlers to notify other managers of the failure of nodes.

Phase 5: **Phase Two.** Once all members have indicated that the Cleanup Phase One has been successfully executed, a second cleanup callback is invoked to allow a coordinated two-phase cleanup. Once all members have signaled the completion of this last cleanup phase they move to the regular operational state and seize the sending of regroup state messages.

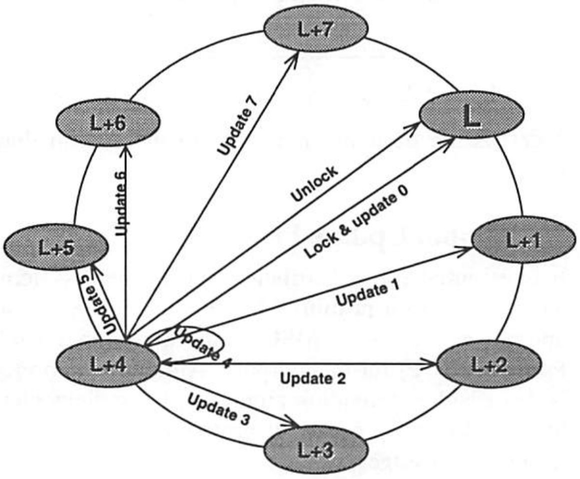


Figure 3. Global Update Sequence

The regroup algorithm in its first two phases is timer driven and the algorithm makes progress independent of the number of nodes in the cluster. The transitions of the next 3 phases are dependent on the number of nodes in the system, but the "information sharing" mechanism makes the system robust in dealing with sporadic message loss.

The state information is broadcast by sending point-to-point datagrams to each node in the cluster. With an inter-transmission period of 300 millisecond, and 31 nodes in the cluster, this generates a background traffic of over 3000 messages/second. A single failure reconfiguration has an average runtime of 3 seconds and thus generates around 10,000 messages. A two-node failure, with a full running cluster is likely to generate between 30,000 and 40,000 messages. Figure

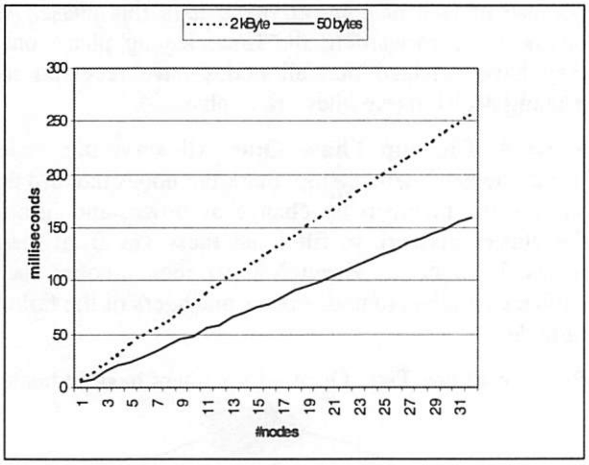


Figure 4. latency of GLUP under ideal conditions

2 details the observed messages in the system during regroup.

7 Global Update Protocol

It is essential for a distributed management system to have access to a primitive that allows consistent state updates at all nodes. MSCS uses the Global Update Protocol (GUP) for this purpose. Although the protocol is described as providing atomicity, its implementation has the stronger property of providing total ordering to its update messages.

When a node starts an global update operation, it first competes for a transmission lock managed by a node that is assigned the functionality of the *locker* node. Only one transmission can be in progress at a time. If the sender can not obtain the lock it is queued on the lock waiting list and blocks until it reaches the head of the queue. With the lock request the sender also transmits its update information to the locker node which applies it locally, and stores the message for later replay under certain failure scenarios. While holding the lock the sender transmits its update to all other active nodes in the cluster and terminates the transmission with a final message to the locker node which releases the lock (see figure 3).

To transmit the messages to all other nodes, the sender organizes the cluster nodes into a circular list, ordered by NodeId. After it acquired the lock the sender send its updates starting with the node that is after the locker node in the list. The sender works through the list in order, wrapping when it reaches the last node in the cluster to the first node and stops when it once again reaches the locker node. The transmission is finished with an unlock message to the locker node.

Acquiring the lock before performing the updates guarantees that only one update is in progress at a given time, which gives the protocol the total ordering property. Atomicity (if one surviving node applies the update, all other surviving nodes will) is achieved through the implementation of a number of fault-handling scenarios.

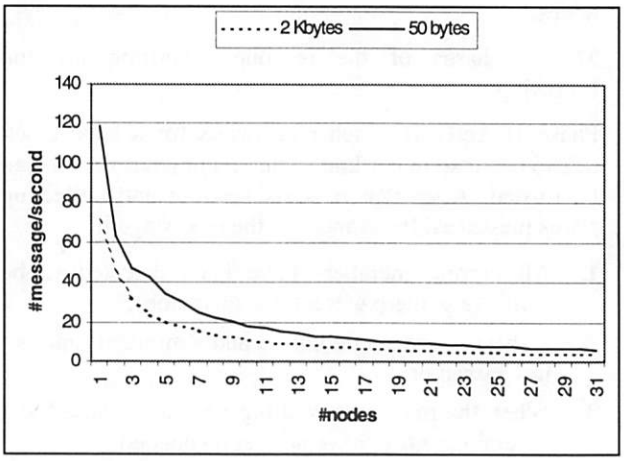


Figure 5. GLUP throughput under ideal conditions

1. *The sender fails*: the locker node takes over the transmission and completes it.
2. *A receiver fails*: wait for the regroup to finish and then finish the transmission.
3. *The locker node fails*: the next node in the node list is assigned locker functionality and the sender treats it as such.
4. *The sender and locker fail*: if the node following the locker has received the update already, in its role as new locker it takes over the transmission.
5. *All nodes that received an update and the sender fail*: pretend the update never happened.

The protocol is implemented as a series of RPC invocations. If an RPC fails, the sender waits for the regroup algorithm to run and install a new membership. GUP will then finish the update series based on the new membership.

Given the strict serial execution of the protocol, its performance is strongly dependent on the number of nodes in the system. The implementation enforces no time bound on the execution of an RPC and any node can introduce unbounded delays as long as RPC keep-alives are being honored.

Repeated measurements show huge variations in results, with the variations being amplified as the number of nodes increases. When a moderate load is placed on the nodes it becomes impossible to produce stable results. These variations can be contributed to the RPC trains, which repeatedly transfer control to the operating system while blocking for the reply. Upon arrival of the reply at the OS level, the Cluster Service needs to compete with other applications that are engaged in IO, to regain CPU control. The non-determinism of the current load state of the system introduces the variances.

The latency of the protocol in an ideal setting is shown in figure 4, the message throughput in figure 5. With 32 nodes the system can handle 6 small (50 bytes) updates/second or 4 larger (2 Kbytes) updates/second.

With systems under a load the protocol breaks down with more then 12 nodes in the cluster. With 10 nodes frequently transmissions are observed that take 2-5 seconds to complete. With 32 nodes transmission times up to one minute were recorded.

8 Discussion

When evaluating the scalability of the distributed components of MSCS it is necessary to separate two issues: the algorithms used and their particular implementation.

8.1 Failure Detection

MSCS is willing to tolerate a long period of silence (7 seconds) before a failure suspicion is raised. This allows for the implementation of mechanisms that can easily deal with large number of nodes. The important scale factor is the number of messages that the nodes need to process both at the sending and the receiving side. Implementing the heartbeat broadcast using repeated point-to-point datagrams does not introduce any problems with 32 nodes, but there is a clear processing penalty at the sender and it will limit the growth to larger numbers.

In an unstructured heartbeat scheme (every node sends heartbeats to all other nodes), the load on the sender and on the network can be significantly reduced by using a true multicast primitive for disseminating the heartbeats. It also removes the sender's dependency on the number of nodes in the system. However, the number of messages a receiver has to process remains proportional to the number of nodes in the system.

More structured approaches have been proposed to reduce the overall complexity of failure detection by imposing a certain structure on the cluster, and localizing failure detection within that structure. A popular approach is to organize the cluster nodes in a logical ring [1,5] where nodes only monitor neighbor nodes in the ring and a token rotates through the ring to disseminate status information. In this scheme however, the token rotation time is dependent on the number of nodes, and the scheme thus has clear scalability limits.

Another aspect of scaling failure detection is the increased chance of multiple concurrent node failures in the cluster. The MSCS mechanism handles multiple failures just as efficient as single failures, while most of the structured failure detection schemes have problems with timely detection of multiple failures and fast reconfiguration of the imposed structure.

Currently the most promising work on failure detection for larger systems is the use of gossip and other epidemic techniques to disseminate availability information [6]. These detectors monitor hundred's of nodes while still providing timely detection, without imposing any significant increased load on nodes and networks.

8.2 Membership Join

The observation that it frequently was impossible to join the 15th or higher node into the cluster is an artifact of the fact that MSCS was not implemented with a large number of nodes in mind. The join reject happens in the phase that is not under control of the sponsor node and where the new node is setting up a mesh of RPC bindings and security contexts with all other active nodes. With 32 nodes this phase is close to a 100 RPC's and any load on the nodes causes significant variations in these serialized executions.

There is no fundamental solution to the problem; if the RPC infrastructure needs to be maintained, the setup phase is needed and some tolerance is needed to allow the mesh to be established. A possible solution would for the joiner to update the sponsor on its progress in this phase to avoid a join rejection.

8.3 Membership Regroup

The membership reconfiguration algorithm works correct under all tested circumstances, independent of the number of nodes used. There are two mechanisms that ensure that the operation performs well, even with a larger number of nodes: (1) The operation is fully distributed, the constant broadcasting of state allows node to rely solely on local observation of global state. (2) The sharing of *"I-have-heard-from-node-X"* information among nodes, makes that the nodes can move to the next phase without having received status messages from all nodes.

Given that a node failure suspicion is not raised until 7 seconds of silence by a node and the first phase of regroup waits for an additional 3 seconds, a problematic node has 10 seconds to recover from some transient failure state. As no false suspicions were ever observed, the timeouts in the first two phases of regroup can be considered to be very conservative. In all observed cases the current membership state was already established well within a second, the remaining time (2-9 seconds) was spent waiting for the failed nodes to respond. As the first phase is dominant in the execution time of the whole regroup operation, a reduction in time can be achieved by combining the failure detection information with the observed regroup state.

A major concern in scaling the regroup operation is the number of messages exchanged. A typical run with 32 nodes generates between 10,000 and 40,000 messages. The status message broadcasts are implemented as series of point-to-point datagrams, which has two major effects: (1) the number of messages generated for the regroup operation grows exponential with the number of nodes and (2) the transmission of 32 identical messages every 300 milliseconds introduces a significant processing overhead at the sender. The regroup algorithm is run at the cluster service, which introduces a user-space/kernel transition for each message, with associated overhead. Introduction of a multicast primitive will allow the implementation to scale at least linearly with the number of nodes and would remove the processing over from the sender of status messages.

8.4 Global Update Protocol

The absence of any concurrency in the message transmission in GUP causes a strict linear increase in latency and decrease in throughput when the number of nodes in the cluster grows.

This serialized and synchronous nature of the protocol is amplified in the particular MSCS implementation. The protocol was originally developed for updating shared OS data-structures, with the update routines running in device interrupt handlers. In MSCS the protocol is implemented uses a series of RPC calls to user-level services. This change in execution environment exposes the vulnerability of the strict serialized operation.

There is no quick solution for the problems that this GUP implementation presents us with. To emulate the original Tandem execution environment the Cluster Service would need to be implemented as a kernel service, which at this point seems impractical.

Replacing GUP with a protocol that provides the same properties but exhibits a more scalable execution style seems preferable. This introduces a number of other complexities, for example many of the currently popular total ordering protocols rely on a tight integration of membership and communication to ensure correct failure handling. This would result in replacing *regroup* as well as GUP.

9 Conclusions

In this paper some of the scalability aspects of the Microsoft Cluster Service were examined. When revisiting the three questions from section 3 the following is concluded:

Can the currently used distributed algorithms be a solid foundation for scalable clusters?

Both failure detection and *regroup* scale well to the numbers that were tested in this paper. When scaling to larger numbers the state processing at receivers will become an issue. The serialized nature of GUP limits its scalability to 10-16 nodes in the current MSCS setup.

Are there any architectural bottlenecks that should be addressed if MSCS needs to be scalable?

The major issue in both failure detection and *regroup* is the implementation of a broadcast facility using repeated point-to-point messages. This introduces a significant overhead on the sender and on the network, and needs to be replaced by a simple multicast primitive. The RPC trains in the membership join operation and in GUP, create a major obstacle for scalability, especially when the systems operate under a significant load.

If MSCS is extended with development support for cluster aware applications are the current distributed services a good basis for these tools?

Support for cluster aware applications has strong requirements in the area of application and component management and failure handling, and requires efficient communication and coordination services. These services would need to be implemented using GUP, which is, in its current form, unsuitable to provide such a service.

To support cluster aware applications a better integration of membership and communication is needed. This will allow for the implementation of a very efficient communication service with properties similar to GUP. Such a service is capable of providing a solid basis for application and component level management and failure handling, and will offer efficient communication and coordination services.

10 Future Work

Research is underway at the Cornell's Reliable Distributed Systems group to investigate and implement alternatives to the distributed management and networking modules in MSCS. Goal is to allow the system to perform well under the scenarios tested for this analysis and to scale to larger numbers (256 nodes and above) at reasonable cost. Recent results such as the scalable failure detection [6] are very promising and show that managing these numbers of nodes is feasible.

In a related project, dubbed Quintet [9,10], new tools are developed to construct highly available, cluster aware application servers. Quintet exploits MSCS features where possible, but at this point provides its own membership and communication modules.

Acknowledgements

Discussions with Jim Gray, Catharine van Ingen, Rod Gamache and Mike Massa have helped to shape the research reported in this paper. The advice of shepherd Ed Lazowska was very much appreciated. Thorsten von Eicken, S. Keshav and Brian Smith graciously contributed hardware to the world's largest *wolfpack* cluster.

References

[1] Badovinatz, P., Chandra, T.D., Gopal, A., Jurgensen, D., Kirby, T., Krishnamur, S., and Pershing, J., "GroupServices: infrastructure for highly available, clustered computing", unpublished document, December 1997

[2] Birman, K.P., *Building Secure and Reliable Network Applications.* Manning Publishing Company, and Prentice Hall, 1997

[3] Carr, R.,"Tandem Global Update Protocol", *Tandem Systems Review*, V1.2 1985.

[4] Katzman., J.A., et.al., "A Fault-tolerant multiprocessor system", United States Patent 4,817,091, March 28, 1989.

[5] Moser, L., Melliar-Smith, M., D. A. Agarwal, D., Budhia, R., and Lingley-Papadopoulos, C., "Totem A Fault-Tolerant Multicast Group Communication System", *Communications of the ACM*, April 1996.

[6] Renesse, R. van, Yaron Minsky, Y., and Hayden, M., "A Gossip-Based Failure Detection Service", in *Proceedings. of Middleware '98*, Lancaster, England, September 1998.

[7] Renesse, R. van, Birman, K., Hayden, M., Vaysburd, A., and Karr, D., "Building Adaptive Systems Using Ensemble", *Software--Practice and Experience*, August 1998.

[8] Vogels, W., Dumitriu, D., Birman, K. Gamache, R., Short, R., Vert, J., Massa, M., Barrera, J., and Gray, J., "The Design and Architecture of the Microsoft Cluster Service -- A Practical Approach to High-Availability and Scalability", *Proceedings of the 28th symposium on Fault-Tolerant Computing,* Munich, Germany, June 1998.

[9] Vogels, W., Dumitriu, D., Panitz, M., Chipalowsky, K., Pettis, J., "Quintet, Tools for Reliable Enterprise Computing", submitted for publication, June 1998.

[10] Vogels, W., van Renesse, R., and Birman, K., "Six Misconceptions about Reliable Distributed Computing", *Proceedings of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998

[11] Vogels, W, "World Wide Failures", *Proceeding of the 1996 ACM SIGOPS Workshop*, Ireland 1996.

Evaluating the Importance of User-Specific Profiling

Zheng Wang
zhwang@eecs.harvard.edu, Division of Engineering and Applied Sciences, Harvard University
Norm Rubin
rubin@ives.amt.tay1.dec.com, Digital Equipment Corporation

Abstract

This paper examines common assumptions about user-specific profiling in profile-based optimization. We study execution profiles of interactive applications on Windows NT to understand how different users use the same program. The profiles were generated by the DIGITAL FX!32 emulator/binary translator system, which automatically runs the x86 version of Windows NT programs on NT/Alpha computers. We have found that people use the benchmark programs in different ways. These differences in program usage can have impact on the performance of profile-based FX!32 program translation and optimization, up to 9%.

1. Introduction

1.1 Background

Profile-based optimization is predicated on the assumption that profiles can be obtained that accurately depict how users run the application. There has been only limited research on the viability of this assumption [FF92, GY96]. We address this problem by examining different users' usage patterns of interactive applications on Windows NT. Here the term "usage pattern" refers to the way a particular individual uses the code in a particular program.

A common assumption in profile-based optimization is that people use applications in similar ways. This assumption is consistent with the behavior of batch-style and computation-intensive programs, and has an implication that user-specific profiling is unnecessary. The alternative to this common assumption is that people use applications in different ways. This is consistent with our intuition for complex and feature-rich programs such as GUI-based interactive applications. It implies that user-specific profiling may be necessary for effective optimization.

The assumption that users are similar or users are different suggests two different models for applying profile-based optimization. In the traditional model, an application is profiled and optimized before its release. Developers run the program with a fixed or arbitrary training workload and use the profile to guide optimizations. Based on the assumption that users are similar, the training workload is considered to be representative. Spike [CG97] is an example of this approach. Some recent systems, such as Morph [ZW97] and FX!32 [HH97], extend the optimization process beyond an application's release by profiling and optimizing the application continuously while it is used[1]. Based on the assumption that users are different, the current versions of Morph and FX!32 operate on a per-user basis. Another assumption in this model is that a particular user's usage pattern may change over time but seldom changes abruptly.

In this study, we compare execution profiles from different users of the same program module. We did not tackle the question of how a particular user's usage pattern changes over time. Since the purpose of the profiles was to guide optimizations, we investigate how the differences in profiles affect optimization performance. We also examine whether we can combine profiles from a group of users to optimize programs for those users and for other users.

Our study shows that users of interactive applications have different usage patterns. For most programs, each individual uses a set of procedures that no other users do, although frequently executed procedures tend to be used by most users. For some benchmarks, profiles from another user or a group can be less effective for optimization than a particular user's own profile.

1.2 Related Work

Although the majority of today's personal computers run mostly interactive applications on Windows systems, there has been little research on how people use these programs. Several research projects investigated Windows operating system performance [CE96, EW96, PS96]. A recent paper [LC98] presented measurements and simulation results of instruction set and architectural characteristics during program execution on x86 Windows NT. These projects focused on the characteristics and comparison of the general system performance, while this paper focuses on the application usage patterns.

[1] Usually, profiling is done continuously when the application is running and optimization is delayed until off-line.

There has been some research on profile comparison for the purpose of branch prediction. Fisher and Freudenberger [FF92] examined the accuracy of predicting conditional branch directions from previous runs of a program. Their experiments focused on batch-computation programs from SPEC benchmark suit, and used subjectively selected datasets to generate profiles. Gloy et al. [GY96] compared user-only traces and full-system traces for dynamic branch prediction. They used standard traces as well as traces from instrumented runs of selected programs. Our profile analysis is aimed for optimization in general, and our profiles were collected from users' unscripted usage of interactive applications.

The next section introduces our experimental methodology, including the collection of the profiles and the statistical analysis methods. Section 3 presents the results, and Section 4 summarizes.

2. Methodology

2.1 FX!32 and FX!32 profiles

We used the DIGITAL FX!32 system to collect execution profiles for Windows NT programs. FX!32 automatically runs x86 applications on Alpha NT, using a combination of emulation and binary translation. When an x86 image is executed under FX!32 for the first time, the FX!32 emulator interprets the x86 code and generates an execution profile. This profile is later used by the FX!32 translator to generate translated and optimized Alpha code. Subsequent executions of the program use the translated code instead of the x86 code. If a certain run of the program uses part of the x86 code that has not been translated, new profile data are generated and merged into the existing profile. The merged profile can be used to re-translate the program.

The contents of the profile reflect program usage over time by a particular user and the addition of new profile data reflects new or changed usage. Therefore, we can learn about the usage patterns of the x86 applications by studying the users' FX!32 profiles. FX!32 profiles are generated during the emulation of x86 code, so they are based on x86 traces, not Alpha traces.

Currently, FX!32 profiles contain information on procedure calls, indirect control transfers, and unaligned memory references. For our statistical analysis of the profiles, we consider only the procedure execution information. In the optimization results study, however, the whole profile is used for the FX!32 program translation/optimization.

One side note is that FX!32's view of program procedures may differ from the set of procedures in the source code. FX!32 works on the binary image and discovers program procedures during emulation. It combines two procedures into a single "FX!32 procedure" if one contains a jump into the other. Therefore, an FX!32 procedure may be the combination of several original procedures. This does not occur frequently, nor does it fundamentally affect our profile analysis. In the rest of the paper, we simply use the term "procedure" to refer to an FX!32 procedure.

2.2 Profile collection

We chose a set of interactive desktop applications as benchmarks for this study. Since FX!32 profiles are generated separately for each program module, we regard each module as a separate benchmark. Different versions of the same program are treated as different modules because they have different code images. For each module, we collected multiple (four or more) profiles, each from a different user.

Our benchmark selection includes executables and DLLs from the Microsoft Office suite as well as other commonly used applications. Table 1 lists the 14 benchmark modules used for this paper. The Office executables and DLLs are noted with their version numbers: 95 (Office Version 7.0 for Windows 95) and 97 (Office 97).

A group of computer system researchers and software developers ran the x86 version of the benchmarks using FX!32 on Alpha computers running Windows NT 4.0. Profiles were generated from their spontaneous and natural usage of the programs. For each module, we collected individual profiles from a selected group of users, each of whom had made significant use of the module[2]. These individual profiles were generated from copies of the module on different machines. Since the machines had comparable hardware and software configuration, differences in the profiles were mostly artifacts of the users' usage patterns and not the execution environment[3].

We calculate a *combined profile* from the individual profiles using the same merging algorithm used by the FX!32 Manager, which sums up the execution counts for each entry. The combined profile represents the

[2] This is evaluated by looking at the profile size, the run count (number of times a module has been executed), and asking the users themselves. The run count alone is not a good indication, because one run of an interactive application may involve a varying number of tasks. Typically, the run count is larger than 10 for these profiles.

[3] As verification, we compared the profiles generated from running an automated script on two machines, and found them virtually identical.

Benchmark Module		Description	Time/Date Stamp	File Size (KB)
MS Office	excel.exe (97)	Office 97 Excel main executable	16:22:31 11/15/96	5469
	mso95.dll (95)	Office 95 (Version 7.0) DLL	01:48:53 07/08/95	918
	mso97.dll (97)	Office 97 DLL	01:02:35 11/07/96	3686
	outllib.dll (97)	Office 97 Microsoft Outlook DLL	20:33:23 11/13/96	4254
	powerpnt.exe (97)	Office 97 PowerPoint executable	05:08:38 11/17/96	3411
	winword.exe (95)	Office 95 Word executable	02:20:46 07/12/95	3755
	winword.exe (97)	Office 97 Word executable	12:33:37 11/15/96	5194
	acrord32.exe	Adobe Acrobat Reader 3.0 executable	16:59:11 06/16/97	2265
	mfc40.dll	Microsoft Visual C++ 4.0 DLL	01:53:24 02/28/96	901
	netscape.exe	Netscape Navigator Gold 3.01 executable	15:42:53 10/22/96	3093
	photoshp.exe	Adobe Photoshop 4.0 executable	09:23:00 10/29/96	3560
	pnui3250.dll	Support library for RealPlayer (32-bit) 5.0	22:54:00 11/22/97	590
	winhlp32.exe	Windows NT 4.0 help utility	14:17:01 07/17/96	303
	winzip32.exe	WinZip compression utility 6.2	17:25:35 10/12/96	846

Table 1. Summary of benchmark modules
Time/Date Stamp is taken from the PE file header.

combined usage of the module by this group of users. We use a series of statistical analyses to examine the similarity between individual profiles and the change in similarity when the profiles grow. We also use the profiles to guide the FX!32 program translation/ optimization and compare the performance of translated programs.

2.3 Statistical analysis

Here we introduce the key methods of our statistical analysis and introduce some terminology that is used in this paper.

An FX!32 profile contains *execution counts* (the number of times a procedure is called) for all procedures that were used during the profile generation. When we compare a group of profiles, we focus on the set of procedures included in each profile, regardless of the procedures' execution counts. This parameter is important for the profile-guided code translation in FX!32 as well as most code layout optimizations. By considering only the set of procedures, we also simplify the comparison and avoid unfair methods of weighing the execution counts. We consider the procedure execution counts only in the part of our analysis that examines the correlation between the number of users who use a procedure and the execution count of the procedure.

We compare the sets of procedures used by individual users by examining their combined profile. If a procedure is included in the combined profile, it has been used by at least one user. For such a procedure, we define its *usage count* as the number of users who have used it. We categorize the procedures in the combined profile according to their usage counts. If a procedure is used by only one user, we call it a *unique procedure.* If it is used by all users, we call it a *common procedure.* Any other procedure is called a *subgroup procedure.*

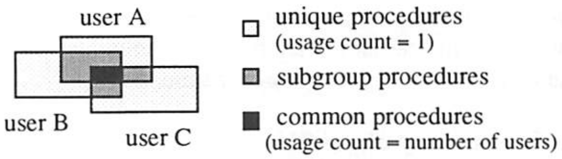


Figure 1. An example of unique, subgroup and common procedures

The usage count distribution of all procedures in the combined profile reflects the similarity between individual profiles. If all users use the same set of procedures, all procedures in the combined profile will be common procedures. If each user uses a different set of procedures, all procedures in the combined profile will be unique procedures. If the sets of procedures used by the individuals are not all the same nor all different, we will see a distribution of unique, subgroup and common procedures in the combined profile. The higher the percentage of common procedures and the lower the percentage of unique procedures, the more similar the individual profiles.

3. Results

In this section, we present a series of statistical analyses of the collected profiles as well as optimization results

Benchmark Module	Number of Users	Number of Procedures in Profile				Smallest %	Largest %	Average %
		Combined	Smallest	Largest	Average			
acrord32.exe	4	5050	4012	4790	4435	79.4%	94.9%	87.8%
excel.exe (97)	4	8514	5885	6821	6351	69.1%	80.1%	74.6%
mfc40.dll	7	2558	1260	1932	1539	49.3%	75.5%	60.2%
mso95.dll (95)	6	2630	1693	2115	1927	64.4%	80.4%	73.3%
mso97.dll (97)	8	9994	5631	8298	6870	56.3%	83.0%	68.7%
netscape.exe	4	7938	4849	7004	5852	61.1%	88.2%	73.7%
outllib.dll (97)	5	16330	10541	13113	11904	64.6%	80.3%	72.9%
photoshp.exe	5	10502	6981	8845	7807	66.5%	84.2%	74.3%
pnui3250.dll	4	1443	1055	1299	1181	73.1%	90.0%	81.8%
powerpnt.exe (97)	5	15014	8905	12895	10504	59.3%	85.9%	70.0%
winhlp32.exe	12	762	543	727	602	71.3%	95.4%	79.0%
winword.exe (95)	5	7317	4600	6222	5398	62.9%	85.0%	73.8%
winword.exe (97)	6	10113	6226	7988	6972	61.6%	79.0%	68.9%
winzip32.exe	5	1125	597	893	737	53.1%	79.4%	65.5%
					Average	63.7%	84.4%	73.2%

Table 2. Summary of profiles

Combined	The number of procedures in the combined profile
Smallest, Largest, Average	The smallest, largest and average number of procedures in an individual profile
Smallest%, Largest%, Average%	"Smallest", "Largest" and "Average" each divided by "Combined"

For every module, each user has used a percentage of the procedures in the combined profile, and *Smallest%*, *Largest%* and *Average%* are the minimum, maximum and average value of this percentage among the group of users.

for two benchmark modules. In our statistical analyses, we examine the similarity between individual profiles and the change in similarity over time.

3.1 Summary of profiles

For every benchmark module, Table 2 lists the number of procedures in the combined profile and the smallest, largest and average number of procedures in an individual profile[4]. Each individual profile reflects one person's usage of the benchmark module over multiple runs, while the combined profile reflects the combined usage by all users of the module. Therefore, each individual profile includes a percentage of the procedures in the combined profile. In Table 2, the lowest *Smallest%* is 49.3% for `mfc40.dll`, which means that one user of `mfc40.dll` has used only 49.3% of all procedures used by the seven users. The highest *Largest%* occurs for `winhlp32.exe`, where one user has used 95.4% of all procedures used by the 12 users. For any module, *Largest%* is 100% if and only if one user has used all procedures used by other users. This does not occur in Table 2, implying that people user the programs in different ways. The average percentage of procedures in the combined profile used by each individual is about 73%.

3.2 Similarity between individual profiles

As discussed in Section 2.3, the usage count distribution of all procedures in the combined profile reflects the similarity between individual profiles. Figure 2 shows the percentage distribution of unique, subgroup and common procedures in the combined profiles for all benchmark modules.

For these benchmark modules, the percentage of common procedures in the combined profile ranges between 38.1% and 76.9%, with an average of 52.1%. The percentage of unique procedures ranges between 7.0% and 23.6%, with an average of 16.4%. In other words, typically about half of the procedures ever used are used by all users, while a small yet significant percentage is used by only one of the users.

Among 14 benchmark modules, `acrord32.exe`, `pnui3250.dll` and `winhlp32.exe` have the highest percentage of common procedures in their

[4] As mentioned in Section 2.1, the term "procedure" refers to an "FX!32 procedure" which may be the combination of several original procedures. The total number of FX!32 procedures in a module is difficult to determine.

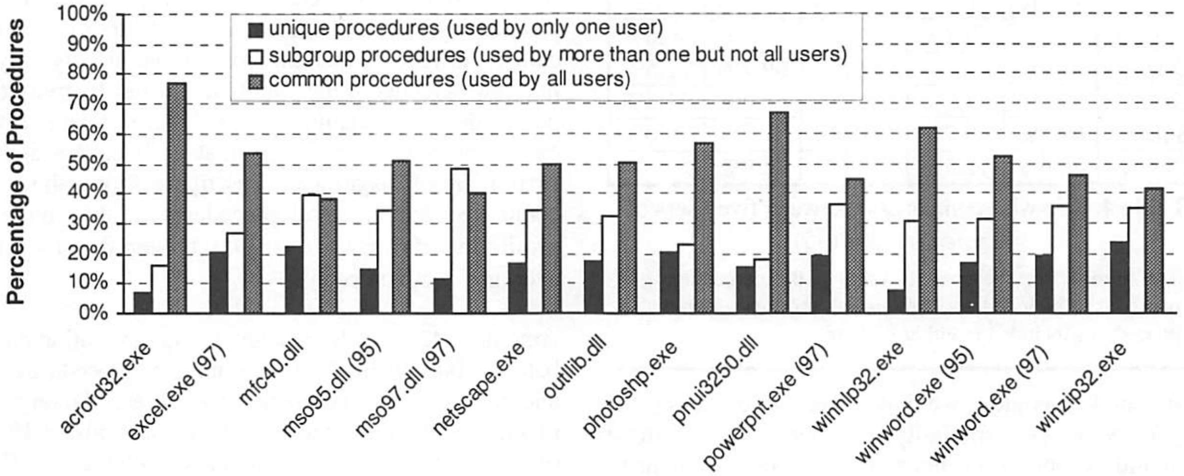


Figure 2. Usage count distribution for all benchmark modules

The Y-axis is the percentage of procedures in the combined profile that fit into a given category. The number of users and the total number of procedures in the combined profile for each benchmark module can be found in Table 2.

combined profiles, and `acrord32.exe` and `winhlp32.exe` also have the lowest percentage of unique procedures. This indicates that each of these three modules shows relatively consistent usage pattern across its users. We notice that these three modules provide less variety of functionality than most other benchmark modules. For example, `acrord32.exe` was mostly used to simply view and print documents downloaded from the Internet. We may also see from Table 1 and Table 2 that `pnui3250.dll` and `winhlp32.exe` are two of the smallest benchmark modules in terms of the file size and the number of procedures. The above two factors may explain the relatively high similarity among each of these three modules' group of individual profiles.

For a more detailed examination, we calculate the distribution of procedure usage counts within each individual profile. Table 3 shows the results for `winword.exe` (95).

We see that every individual profile has its share of unique procedures and subgroup procedures. The common procedures constitute between 61.4% and 83.1% of the procedures in an individual profile, while the percentage of unique procedures in an individual profile ranges from 1.2% to 9.0%. In terms of the procedures included, none of the profiles is a subset or superset of another profile. We have observed similar phenomena for other benchmark modules. For several benchmark modules, one or more relatively small individual profiles have no unique procedures, but they still contain subgroup procedures. `winhlp32.exe` and `netscape301.exe` are the only two benchmark modules where one person uses a subset of the procedures another person uses.

Profiled User	# of Proc.	Number of Proc. by Usage Count		
		1 (*unique*)	2-4 (*subgroup*)	5 (*common*)
Bashful	4600	55	722	3823
Doc	4990	69	1098	3823
Grumpy	5332	210	1299	3823
Sneezy	5846	312	1711	3823
Happy	6222	562	1837	3823
Combined	7317	1208	2286	3823

Table 3. Procedure distribution among five users of `winword.exe` (95)

# of Proc.: number of procedures in the profile

We have replaced all user names with pseudonyms. For each user, unique procedures are those used by this user but none of the other four. Common procedures are those used by all five users.

We also examine whether there are highly similar usage patterns among small groups of users. To evaluate this, we use pair-wise comparison between all individual profiles for a module to see whether some of them have significantly higher similarity among themselves than with other profiles. For each pair of profiles, we calculate the percentage of procedures included in both among all procedures included in either of them. This percentage measures the similarity between two profiles. Table 4 lists the results for `winword.exe` (95). All numbers in the table fall between 66.6% and 77.2%, indicating that similarity between each pair of users is on a close level. In the analysis for other

	Bashful	Doc	Grumpy	Sneezy	Happy
Bashful	--	77.2%	69.6%	69.1%	66.6%
Doc	77.2%	--	73.5%	69.7%	72.5%
Grumpy	69.6%	73.5%	--	76.4%	71.6%
Sneezy	69.1%	69.7%	76.4%	--	76.0%
Happy	66.6%	72.5%	71.6%	76.0%	--

Table 4. Pair-wise comparison between five users of `winword.exe` (95)

The number for each pair of users is the percentage of procedures included in both users' profiles among all procedures included in either of them.

benchmark modules, we have seen a few cases of relatively higher similarity between two or three individuals, but we do not think they are sufficient to conclude that there is particularly high similarity among small groups of users.

Results in this subsection imply that users use applications in different ways, supporting the theory that user-specific profiling is important for effective optimization.

3.3 Correlation between procedure usage count and execution count

In this subsection, we examine whether there is correlation between a procedure's usage count and its execution count. In many profile-based optimizations, priority is given to the most frequently executed code. In this case, procedures with higher execution counts are more important to the optimization than those with lower counts.

The procedure execution counts in FX!32 profiles do not always match the traditional definition of procedure execution count. In FX!32, control transfers within the translated Alpha code are not captured in the profile. Since a user may perform program translation from time to time, a procedure's execution count in the profile may be less than the number of times it has been called. However, experience shows that usually only up to a few percents of the counts will differ by more than one order of magnitude. For Figure 3, we divide execution counts into levels that each covers at least two orders of magnitude. This figure shows the usage count distribution of all procedures in the combined profile for `winword.exe` (95), broken down by their average execution counts.

Among 3842 procedures with average execution counts below 100, about 30% are unique procedures and another 30% are common procedures. Among 724 procedures with average execution counts above 10000, over 90% are used by all or all but one users. These numbers show that frequently executed procedures are more likely to appear in all or most individual profiles than those executed less frequently. In other words, the part of code that users execute the most is similar, despite the differences in their overall profiles. The statistics for other benchmark modules also support this conclusion.

For optimizations that give priority to frequently executed code, this conclusion on similarity suggests that despite the differences among users, we may find a representative training workload that exercises the procedures frequently used by most users. On the other hand, such a workload may not cover enough procedures for any one user. For `winword.exe` (95), 2535 procedures have been used by four or five users with average execution counts above 100, while the smallest individual profile includes 4600 procedures and the combined profile includes 7317 procedures. Further analysis in the context of a specific optimization will determine the tradeoff between using such a user-independent training workload and using user-specific profiling.

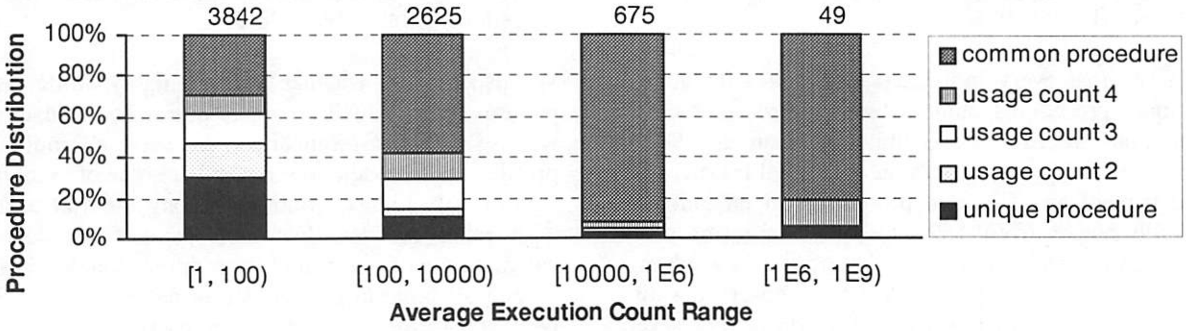


Figure 3. Procedure distribution: usage count vs. execution count: `winword.exe` (95), five users

Average Execution Count: procedure execution count averaged over users who have use this procedure
(Average Execution Count = execution count in combined profile / usage count)
number at the top of each bar: the total number of procedures in this range

3.4 Change in similarity when profiles grow

When the individual profiles grow larger with more use of the program, one might speculate that their similarity increases as they all approach one common limit, the set of all procedures in the program. Table 5 examines the change in similarity between five `winword.exe` (95) profiles when some of them grow larger.

We took snapshots of the five individual profiles at six different times during a month. Each time at least one profile had grown since the last time. During the first five snapshots, the percentage of common procedures in the combined profile increased and the percentage of unique procedures decreased. In these cases, the similarity between individual profiles increased when some of them grew larger. However, the slow change in the similarity suggests that the individual profiles might never "converge"; that is, profiles from different users may never reach a certain high level of similarity, reflecting their different usage patterns. In fact, in the last snapshot where *Sneezy*'s profile grew, the similarity between individual profiles slightly decreased due to the new procedures *Sneezy* had started to use. In summary, the users' accumulated usage patterns may become more similar with more use of the program, but some differences persist.

3.5 Optimization performance

This subsection examines the impact of differences in profiles on the performance of programs optimized using the profiles. This impact is dependent on how the profile information is used during the optimization. Different optimizations may have varying sensitivity to differences in profiles. Even with the same optimization, the performance impact may vary for different programs and different workloads.

In our experiments, the FX!32 translator/optimizer uses the profile to determine the set of code to translate and to guide common compiler optimizations, such as procedure layout, procedure inlining and dead code elimination, on the translated code. We used different training profiles, both individual and combined, to translate/optimize the same module. For each profile, we measured the performance of the application using the translated code. Based on the results we discuss the effectiveness of using profiles from another user or a group of users to translate/optimize a program.

One difficulty in our experiments was measuring the performance of an interactive application. To achieve repeatability, we chose to measure the execution time of a standard, script-driven workload. We consequently assumed that our "test user" performed this same workload for both training and testing. In reality, a user's usage pattern of a program may change over time, causing the testing workload to be different from the training workload. The impact of this factor on program optimization is not investigated in this paper.

We conducted our experiments on benchmark modules `winword.exe` (95) and `powerpnt.exe` (97).

3.5.1 Microsoft Word benchmark

For testing on `winword.exe` (95) from Microsoft Word 7.0 for Windows 95, we used the workload from SYSmark32 for Windows NT version 1.0 distributed by BAPCo [BAPCo]. We included two individual profiles used in the statistical analysis from users *Grumpy* and *Happy*, plus two relatively small individual profiles *Dopey* and *Sleepy* to examine the issue of "under-training". Figure 4 shows the results for these four individual profiles and various combined profiles. All performance measurements were done on a 500MHz Alpha computer with 64MB of main memory. Word 7.0 was the only application running on the system.

Without any translation/optimization, the program is executed entirely through emulation, which is slow (459 seconds). The *Minimal* profile, generated by starting up `winword.exe` and exiting immediately, is practically the smallest user profile possible and a subset of any real user profile. Its test result (292 seconds) indicates a lower bound of optimization benefit one should expect from using any profile.

	Number of Procedures Used By						Procedure Distribution by Usage Count		
Date	Bashful	Doc	Grumpy	Sneezy	Happy	*Combined*	1 (*unique*)	2-4 (*subgroup*)	5 (*common*)
10/10/97	4600	4091	4691	5648	6222	7191	20.4%	33.5%	46.2%
10/15/97	4600	*4465	4691	5648	6222	7213	19.5%	33.2%	47.3%
10/22/97	4600	4465	*4947	5648	6222	7239	18.2%	34.0%	47.8%
10/29/97	4600	*4834	*5332	5648	6222	7283	17.0%	31.6%	51.8%
11/03/97	4600	*4990	5332	5648	6222	7288	16.3%	31.5%	52.3%
11/10/97	4600	4990	5332	*5846	6222	7317	16.5%	31.2%	52.2%

* A profile that grew

Table 5. Change in similarity when individual profiles grow larger: `winword.exe` (95), five users

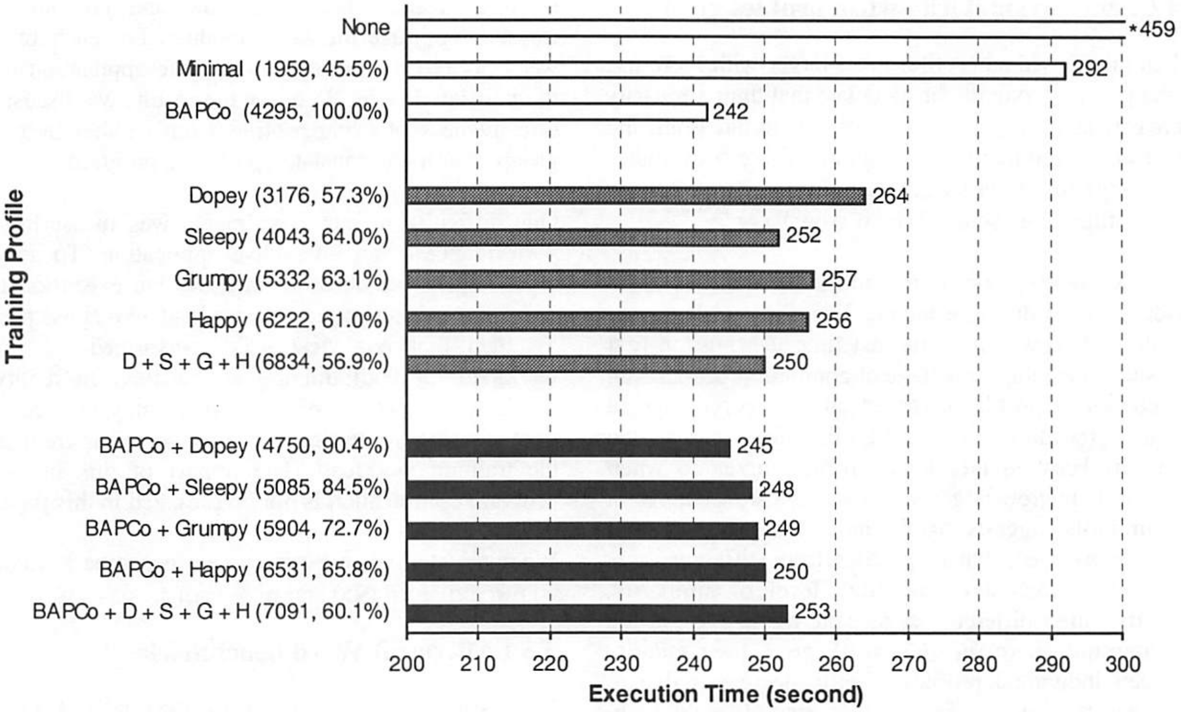


Figure 4. Optimization results for `winword.exe` (95)

Training Profile The profile used to translate/optimize the program

Execution Time The execution time of the BAPCo workload using the translated code. Average of three warm runs. Standard deviation is within 2 seconds for all numbers except 6 seconds for "None"

(number, %) The number of procedures in the training profile, the similarity between the training profile and the BAPCo profile (calculated as the percentage of procedures included in both profiles among all procedures included in either of them)

None No profile, and therefore no translation/optimization

Minimal profile The profile generated by starting up `winword.exe` and then exiting immediately

BAPCo profile The profile generated from one run of the testing workload from BAPCo

+ The combining of profiles

D, S, G, H Dopey, Sleepy, Grumpy, Happy

We draw several conclusions from Figure 4:

1. We achieve the best performance (242 seconds) when we translate the program by using the test user profile. When using a profile from another user or a group, we see performance that is 1–9% worse (245-264 seconds) but still much better than using the *Minimal* profile.

2. The optimization benefit has a rough trend of increasing with the similarity between the training profile and the testing profile. However, this relation is not monotonic.

3. In the graph, the black bars correspond to profiles from groups that include the test user, while the gray bars correspond to profiles from other users and groups that do not include the test user. With the exception of *BAPCO+Dopey+Sleepy+Grumpy+Happy*, "black bar profiles" provide more effective optimization than "gray bar profiles." This suggests that a group's combined profile is more effective for optimization if the group includes the test user than if not.

4. Among combined profiles that include the test user profile (the black bars), the larger the profile, the less the optimization benefit. This suggests that a combined profile may become less effective for a user in the group when the group is large. A possible explanation is that extra procedures in the translated code increase memory system load and cause sub-optimal code layout. This may also explain why *BAPCO+Dopey+Sleepy+Grumpy+Happy* provides less effective optimization than *Dopey+Sleepy+Grumpy+Happy.*

For this benchmark, user-specific profiling has measurable impact on optimization performance.

3.5.2 Microsoft PowerPoint benchmark

For `powerpnt.exe` (97) from Microsoft PowerPoint 97, we used an automated testing script designed at Digital Equipment Corporation. Similar with BAPCo workloads, it uses Microsoft Visual Test to drive the application. This script was originally designed to test the functionality of PowerPoint. It included some wait time in between tasks. In this sense, it may be closer to a real user than BAPCo workloads, which mostly consist of continuously executed CPU-intensive tasks. On the other hand, some application response time and background activity may be hidden by the wait time, making the throughput measurement of execution time less sensitive to the code quality. Figure 5 shows the results. All performance measurements were done on a 500MHz Alpha computer with 128MB of main memory. PowerPoint 97 was the only application running on the system.

The results show that all the individual and combined profiles are almost equally effective for optimization, with differences on the level of 1%. This implies that for this program and this workload, user-specific profiling does not have significant impact on the performance of FX!32 translation/optimization.

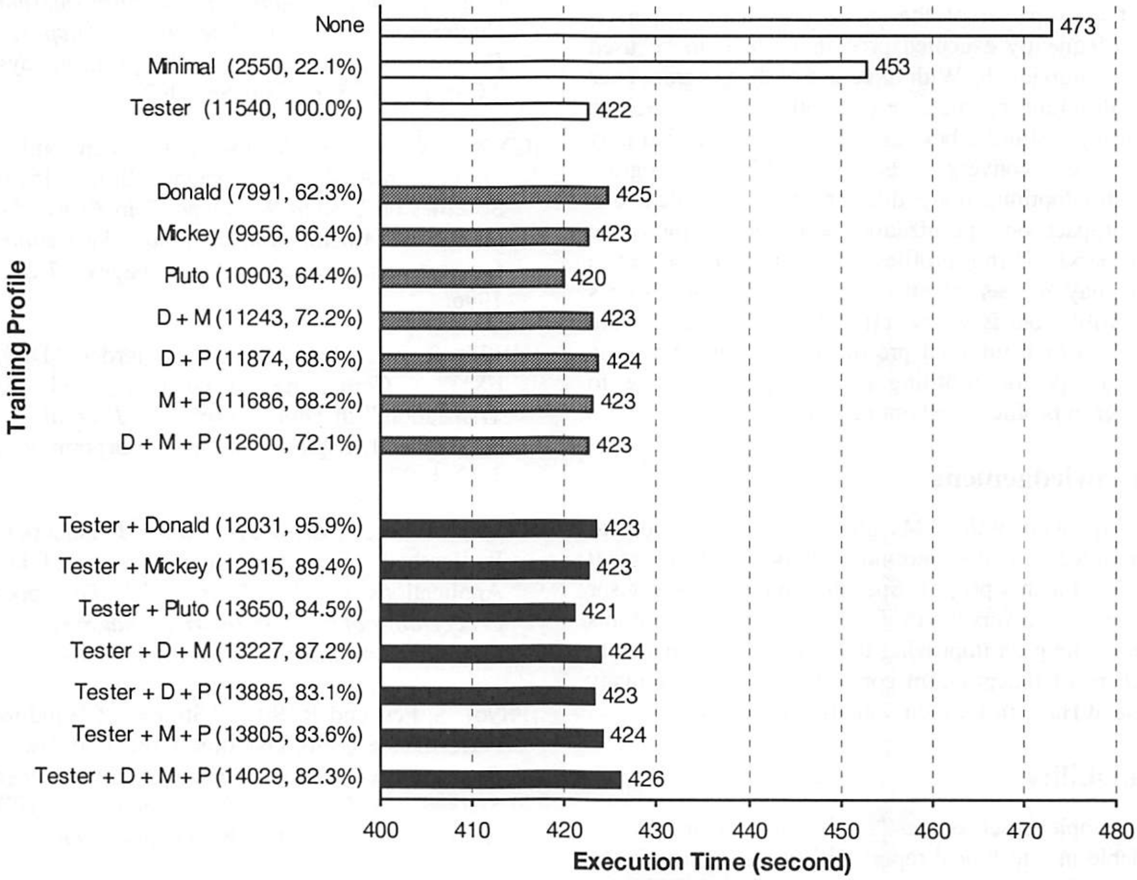


Figure 5. Optimization results for `powerpnt.exe` (97)

Training Profile — The profile used to translate/optimize the program

Execution Time — The execution time of the testing workload using the translated code. Average of three warm runs. Standard deviation is within 2 seconds for all numbers

(number, %) — The number of procedures in the training profile, the similarity between the training profile and the Tester profile (calculated as the percentage of procedures included in both profiles among all procedures included in either of them)

None — No profile, and therefore no translation/optimization

Minimal profile — The profile generated by starting up `powerpnt.exe` and then exiting immediately

Tester profile — The profile generated from one run of the testing script

+ — The combining of profiles

D, M, P — Donald, Mickey, Pluto

Results for these two benchmarks indicate that depending on the program and the workload, differences in profiles can have measurable or insignificant impact on optimization performance.

4. Summary

This paper has compared and analyzed FX!32 profiles from different users for a set of Windows NT programs. We discovered that the sets of procedures used by individuals are fairly different. Among all procedures used by a group of users, typically around 50% are used by all users, while 7-24% are used by only one of the users. In most cases, the users have usage patterns different from each other, without anyone using a subset or superset of the procedures another person uses. Frequently executed procedures tend to be used by most individuals. With more use of the program over time, different people's usage patterns may become increasingly similar, but our results suggested that they will never converge. For the FX!32 program translation/optimization, differences in profiles can have impact on optimization performance for some benchmarks. Using profiles from another user or a group may be less effective than using the test user's own profile, but is always effective compared to using no profile or a minimal profile. Overall, we conclude that user-specific profiling is an important factor to consider in profile-based optimization.

Acknowledgement

Many members of the AMT group at Digital Equipment Corporation provided enormous support and important feedback for this project. Special thanks to my advisor at Harvard University, Prof. J. Bradley Chen, for his generous help on improving this paper. Also, thanks to members of the program committee as well as many people at Harvard for their valuable comments.

Availability

The complete set of results for all benchmarks is available in a technical report [WR98] and through the URL http://www.eecs.harvard.edu/~zhwang/NT98/

References

[BAPCo] Business Applications Performance Corporation, http://www.bapco.com/

[CE96] J. B. Chen, Y. Endo, K. Chan, D. Mazieres, A. Dias, M. Seltzer, and M. D. Smith, "The Measured Performance of Personal Computer Operating Systems." In *ACM Transactions on Computer Systems* 14:1, pages 3-40, February 1996.

[CG97] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin, "Spike: An Optimizer for Alpha/NT Executables." In *Proceedings of the USENIX Windows NT Workshop*, USENIX Association, pages 17-24, August 1997.

[EW96] Y. Endo, Z. Wang, J. B. Chen, and M. I. Seltzer, "Using Latency to Evaluate Interactive System Performance." In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, USENIX Association, pages 185-199, October 1996.

[FF92] J. Fisher and S. Freudenberger, "Predicting Conditional Branches from Previous Runs of a Program." In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, pages 85-95, October 1992.

[GY96] N. Gloy, C. Young, J. B. Chen, and M. D. Smith, "An Analysis of Dynamic Branch Prediction Schemes on System Workloads." In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ACM, pages 12-21, May 1996.

[HH97] R. J. Hookway and M. A. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation." In *Digital Technical Journal*, Volume 9, Number 1, Digital Equipment Corporation, pages 3-12, 1997.

[LC98] D. Lee, P. Crowley, J. Baer, T. Anderson, and B. Bershad, "Execution Characteristics of Desktop Applications on Windows NT." To appear in *Proceedings of the 25th International Symposium on Computer Architecture*, IEEE, June 1998.

[PS96] S. Perl and R. Sites, "Studies of Windows NT Performance Using Dynamic Execution Traces." In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, USENIX Association, pages 169-183, October 1996.

[WR98] Z. Wang and N. Rubin, "A Statistical Analysis of User-Specific Profiles." Technical Report TR-09-98, Computer Science Group, Harvard University, July 1998.

[ZW97] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System Support for Automated Profiling and Optimization." In *Proceedings of the 16th ACM Symposium of Operating Systems Principles*, ACM, pages 15-26, October 1997.

Cygwin32: A Free Win32 Porting Layer for UNIX® Applications

Geoffrey J. Noer
noer@cygnus.com

Cygnus Solutions
1325 Chesapeake Terrace
Sunnyvale, CA 94089

Abstract

Cygwin32 is a full-featured Win32 porting layer for UNIX applications, compatible with all Win32 hosts (currently Microsoft Windows NT, Windows 95, and Windows 98). It was invented in 1995 by Cygnus Solutions as part of the answer to the question of how to port the GNU development tools to the Win32 host.

The Win32-hosted GNUPro compiler tools that use the library are available for a variety of embedded processors as well as a native version for writing Win32 programs. By basing this technology on the GNU tools, Cygnus provides developers with a high-performance, feature-rich 32-bit code development environment, including a graphical source-level debugger.

Cygwin32 is a Dynamic-Linked Library (DLL) that provides a large subset of the system calls found in common UNIX implementations. The current release includes all POSIX.1/90 calls except for `setuid` and `mkfifo`, all ANSI C standard calls, and many common BSD and SVR4 services including Berkeley sockets.

This article will discuss our experiences porting the GNU development tools to the Win32 host and explore the development and architecture of the Cygwin32 library.

1. Introduction

Cygnus Solutions was founded in 1989 to provide commercial support and development services for the GNU development tools, focusing on the embedded computing industry. The tools include the GNU C/C++ compiler (GCC/G++), assembler (GAS), linker (GLD), and debugger (GDB). As of June 1998, Cygnus sells support for over 150 host/target combinations.

When the Free Software Foundation (FSF) first wrote the GNU tools in the mid-1980s, portability among existing and future UNIX operating systems was an important goal. By mid-1995, the tools had been ported to 16-bit DOS using the go32 32-bit extender by D.J. Delorie[1]. However, no one had completed a native 32-bit port for Windows NT and 95/98. It seemed likely that the demand for Win32-hosted native and cross-development tools would soon be large enough to justify the development costs involved.

2. Porting the GNU Compiler to Win32

The first step in porting the compiler tools to Win32 was to enhance them so that they could generate and interpret Win32 native object files, using Microsoft's Portable Executable (PE) format. This proved to be relatively straightforward because of similarities to the Common Object File Format (COFF), which the GNU tools already supported. Most of these changes were confined to the Binary File Descriptor (BFD) library and to the linker.

In order to support the Win32 Application Programming Interface (API), we extended the capabilities of the binary utilities to handle Dynamic-Linked Libraries (DLLs). After creating export lists for the specific Win32 API DLLs that are shipped with Win32 hosts, the tools were able to generate static libraries that executables could use to gain access to Win32 API functions. Because of redistribution restrictions on Microsoft's Win32 API header files, we wrote our own Win32 header files from scratch on an as-needed basis. Once this work was completed, we were able to build UNIX-hosted cross-compilers capable of generating valid PE executables that ran on Win32 systems.

The next task was to port the compiler tools themselves to Win32. Previous experiences using Microsoft Visual C++ to port GDB convinced us to find another means for bootstrapping the full set of tools. In addition to wanting to use our own compiler technology, we wanted a portable build system. The GNU development tools' configuration and build procedures require a large number of additional UNIX utilities not available on Win32 hosts. So we decided to use UNIX-hosted cross-compilers to build our Win32-hosted native and cross-development tools. It made perfect sense to do this since we were successfully using a nearly identical technique to build our DOS-hosted products.

The next obstacle to overcome was the many dependencies on UNIX system calls in the sources, especially in the GNU debugger GDB. While we could have rewritten sizable portions of the source code to work within the context of the Win32 API (as was done for the DOS-hosted tools), this would have been prohibitively time-consuming. Worse, we would have introduced conditionalized code that would have been expensive to maintain in the long run. Instead, Cygnus developers took a substantially different approach by writing Cygwin32.

3. Initial Goals

The original goal of Cygwin32 was simply to get the development tools working. Completeness with respect to POSIX.1[2] and other relevant UNIX standards was not a priority.

Part of our definition of "working native tools" is having a build environment similar enough to UNIX to support rebuilding the tools themselves on the host system, a process we call self-hosting. The typical configuration procedure for a GNU tool involves running "configure", a complex Bourne shell script that determines information about the host system. The script then uses that information to generate the Makefiles used to build the tool on the host in question.

This configuration mechanism is needed under UNIX because of the large number of varying flavors of UNIX. If Microsoft continues to produce new variants of the Win32 API as it releases new versions of its operating systems, it may prove to be quite valuable on the Win32 host as well.

The need to support this configuration procedure added the requirement of supporting user tools such as sh, make, file utilities (e.g. ls and rm), text utilities (e.g. cat, tr), shell utilities (e.g. echo, date, uname), sed, awk, find, xargs, tar, and gzip, among many others. Previously, most of these user tools had only been built natively (on the host on which they would run). As a result, we had to modify their configure scripts to be compatible with cross-compilation.

Other than making the necessary configuration changes, we wanted to avoid Win32-specific changes since the UNIX compatibility was to be provided by Cygwin32 as much as possible. While we knew this would be a sizable amount of work, there was more to gain than just achieving self-hosting of the tools. Supporting the configuration of the development tools would also provide an excellent method of testing the Cygwin32 library.

Although we were able to build working Win32-hosted toolchains with cross-compilers relatively soon after the birth of Cygwin32, it took much longer than we expected before the tools could reliably rebuild themselves on the Win32 host because of the many complexities involved.

4. "Harnessing the Power of the Internet"

Instead of keeping the Cygwin32 technology proprietary and developing it in-house, Cygnus chose to make it publicly available under the terms of the GNU General Public License (GPL), the traditional license for the GNU tools. Since its inception, we have made a new "GNU-Win32 beta release" available via ftp over the Internet every three or four months. Each release includes binaries of Cygwin32 and the development tools, coupled with the source code needed to build them. Unlike standard Cygnus products, these free releases come without any assurances of quality or support, although we provide a mailing list that is used for discussion and feedback.

In retrospect, making the technology freely available was a good decision because of the high demand for quality 32-bit native tools in the Win32 arena, as well as significant additional interest in a UNIX portability layer like Cygwin32. While far from perfect, the beta releases are good enough for many people. They provide us with tens of thousands of interested developers who are willing to use and test the tools. A few of them are even willing to contribute code fixes and new functionality to the library. As of the last public release, developers on the Net had written or improved a significant portion of the library, including important aspects such as support for UNIX signals and the TTY/PTY calls.

In order to spur as much Net participation as possible, the Cygwin32 project features an open development model. We make weekly source snapshots available to the general public in addition to the periodic full GNU-Win32 releases. A developers' mailing list facilitates discussion of proposed changes to the library.

In addition to the GPL version of Cygwin32, Cygnus provides a commercial license for supported customers of the native Win32 GNUPro tools.

5. The Cygwin32 Architecture

Now we turn to an analysis of the actual architecture of the Cygwin32 library.

When a binary linked against the library is executed, the Cygwin32 DLL is loaded into the application's text segment. Because we are trying to emulate a UNIX kernel which needs access to all processes running under it, the first Cygwin32 DLL to run creates shared memory areas that other processes using separate instances of the DLL can access. This is used to keep track of open file descriptors and assist `fork` and `exec`, among other purposes. In addition to the shared memory regions, every process also has a `per_process` structure that contains information such as process id, user id, signal masks, and other similar process-specific information.

The DLL is implemented using the Win32 API, which allows it to run on all Win32 hosts. Because processes run under the standard Win32 subsystem, they can access both the UNIX compatibility calls provided by Cygwin32 as well as any of the Win32 API calls. This gives the programmer complete flexibility in designing the structure of their program in terms of the APIs used. For example, they could write a Win32-specific GUI using Win32 API calls on top of a UNIX back-end that uses Cygwin32.

Early on in the development process, we made the important design decision that it would not be necessary to strictly adhere to existing UNIX standards like POSIX.1 if it was not possible or if it would significantly diminish the usability of the tools on the Win32 platform. In many cases, an environment variable can be set to override the default behavior and force standards compliance.

5.1. Windows NT != Windows 95/98

While Windows 95 and Windows 98 are similar enough to each other that we can safely ignore the distinction when implementing Cygwin32, Windows NT is an extremely different operating system. For this reason, whenever the DLL is loaded, the library checks which operating system is active so that it can act accordingly.

In some cases, the Win32 API is only different for historical reasons. In this situation, the same basic functionality is available under 95/98 and NT but the method used to gain this functionality differs. A trivial example: in our implementation of `uname`, the library examines the `sysinfo.dwProcessorType` structure member to figure out the processor type under 95/98. This field is not supported in NT, which has its own operating system-specific structure member called `sysinfo.wProcessorLevel`.

Other differences between NT and 95/98 are much more fundamental in nature. The best example is that only NT provides a security model.

5.2. Permissions and Security

Windows NT includes a sophisticated security model based on Access Control Lists (ACLs). Although some modern UNIX operating systems include support for ACLs, Cygwin32 maps Win32 file ownership and permissions to the more standard, older UNIX model. The `chmod` call maps UNIX-style permissions back to the Win32 equivalents. Because many programs expect to be able to find the `/etc/passwd` and `/etc/group` files, we provide utilities that can be used to construct them from the user and group information provided by the operating system.

Under Windows NT, the administrator is permitted to `chown` files. There is currently no mechanism to support the `setuid` concept or API call. Although we hope to support this functionality at some point in the future, in practice, the programs we have ported have not needed it.

Under Windows 95/98, the situation is considerably different. Since a security model is not provided, Cygwin32 fakes file ownership by making all files look like they are owned by a default user and group id. As under NT, file permissions can still be determined by examining their read/write/execute status. Rather than return an unimplemented error, under Windows 95/98, the `chown` call succeeds immediately without actually performing any action whatsoever. This is appropriate since essentially all users jointly own the files when no concept of file ownership exists.

It is important that we discuss the implications of our "kernel" using shared memory areas to store information about Cygwin32 processes. Because these areas are not yet protected in any way, in principle a malicious user could modify them to cause unexpected behavior in Cygwin32 processes. While this is not a new problem under Windows 95/98 (because of the lack of operating system security), it does constitute a security hole under Windows NT. This is because one user could affect the Cygwin32 programs run by another user by changing the shared memory information in ways that they could not in a more typical WinNT program. For this reason, it is not appropriate to use Cygwin32 in high-security applications. In practice, this will not be a major problem for most uses of the library.

5.3. Files

Cygwin32 supports both Win32- and POSIX-style paths, using either forward or back slashes as the directory delimiter. Paths coming into the DLL are translated from Win32 to POSIX as needed. As a result, the library believes that the file system is a POSIX-compliant one, translating paths back to Win32 paths whenever it calls a Win32 API function. UNC pathnames (starting with two slashes) are supported.

The layout of this POSIX view of the Windows file system space is stored in the Windows registry. While the slash ('/') directory points to the system partition by default, this is easy to change with the Cygwin32 `mount` utility. In addition to selecting the slash partition, it allows mounting arbitrary Win32 paths into the POSIX file system space. Many people use the utility to mount each drive letter under the slash partition (e.g. `C:\` to `/c`, `D:\` to `/d`, etc...).

The library exports several Cygwin32-specific functions that can be used by external programs to convert a path or path list from Win32 to POSIX or vice versa. Shell scripts and Makefiles cannot call these functions directly. Instead, they can do the same path translations by executing the "cygpath" utility program that we provide with Cygwin32.

Win32 file systems are case preserving but case insensitive. Cygwin32 does not currently support case distinction because, in practice, few UNIX programs actually rely on it. While we could mangle file names to support case distinction, this would add unnecessary overhead to the library and make it more difficult for non-Cygwin32 applications to access those files.

Symbolic links are emulated by files containing a magic cookie followed by the path to which the link points. They are marked with the System attribute so that only files with that attribute have to be read to determine whether or not the file is a symbolic link. Hard links are fully supported under Windows NT on NTFS file systems. On a FAT file system, the call falls back to simply copying the file, a strategy that works in many cases.

The inode number for a file is calculated by hashing its full Win32 path. The inode number generated by the `stat` call always matches the one returned in `d_ino` of the `dirent` structure. It is worth noting that the number produced by this method is not guaranteed to be unique. However, we have not found this to be a significant problem because of the low probability of generating a duplicate inode number.

5.4. Text Mode vs. Binary Mode

Interoperability with other Win32 programs such as text editors was critical to the success of the port of the development tools. Most Cygnus customers upgrading from the older DOS-hosted toolchains expected the new Win32-hosted ones to continue to work with their old development sources.

Unfortunately, UNIX and Win32 use different end-of-line terminators in text files. Consequently, carriage-return newlines have to be translated on the fly by Cygwin32 into a single newline when reading in text mode. The control-z character is interpreted as a valid end-of-file character for a similar reason.

This solution addresses the compatibility requirement at the expense of violating the POSIX standard that states that text and binary mode will be identical. Consequently, processes that attempt to `lseek` through text files can no longer rely on the number of bytes read as an accurate indicator of position in the file. For this reason, an environment variable can be set to override this behavior.

5.5. ANSI C Library

We chose to include Cygnus' own existing ANSI C[3] library "newlib" as part of the library, rather than write all of the lib C and math calls from scratch. Newlib is a BSD-derived ANSI C library, previously only used by cross-compilers for embedded systems development.

The reuse of existing free implementations of such things as the `glob`, `regexp`, and `getopt` libraries saved us considerable effort. In addition, Cygwin32 uses Doug Lea's free `malloc` implementation that successfully balances speed and compactness. The library accesses the `malloc` calls via an exported function pointer. This makes it possible for a Cygwin32 process to provide its own malloc if it so desires.

5.6. Process Creation

The `fork` call in Cygwin32 is particularly interesting because it does not map well on top of the Win32 API. This makes it very difficult to implement correctly. Currently, the Cygwin32 `fork` is a non-copy-on-write implementation similar to what was present in early flavors of UNIX.

The first thing that happens when a parent process forks a child process is that the parent initializes a space in the Cygwin32 process table for the child. It then creates a suspended child process using the Win32

CreateProcess call. Next, the parent process calls setjmp to save its own context and sets a pointer to this in a Cygwin32 shared memory area (shared among all Cygwin32 tasks). It then fills in the child's .data and .bss sections by copying from its own address space into the suspended child's address space. After the child's address space is initialized, the child is run while the parent waits on a mutex. The child discovers it has been forked and longjumps using the saved jump buffer. The child then sets the mutex the parent is waiting on and blocks on another mutex. This is the signal for the parent to copy its stack and heap into the child, after which it releases the mutex the child is waiting on and returns from the fork call. Finally, the child wakes from blocking on the last mutex, recreates any memory-mapped areas passed to it via the shared area, and returns from fork itself.

While we have some ideas as to how to speed up our fork implementation by reducing the number of context switches between the parent and child process, fork will almost certainly always be inefficient under Win32. Fortunately, in most circumstances the spawn family of calls provided by Cygwin32 can be substituted for a fork/exec pair with only a little effort. These calls map cleanly on top of the Win32 API. As a result, they are much more efficient. Changing the compiler's driver program to call spawn instead of fork was a trivial change and increased compilation speeds by twenty to thirty percent in our tests.

However, spawn and exec present their own set of difficulties. Because there is no way to do an actual exec under Win32, Cygwin32 has to invent its own Process IDs (PIDs). As a result, when a process performs multiple exec calls, there will be multiple Windows PIDs associated with a single Cygwin32 PID. In some cases, stubs of each of these Win32 processes may linger, waiting for their exec'd Cygwin32 process to exit.

5.7. Signals

When a Cygwin32 process starts, the library starts a secondary thread for use in signal handling. This thread waits for Windows events used to pass signals to the process. When a process notices it has a signal, it scans its signal bitmask and handles the signal in the appropriate fashion.

Several complications in the implementation arise from the fact that the signal handler operates in the same address space as the executing program. The immediate consequence is that Cygwin32 system functions are interruptible unless special care is taken to avoid this. We go to some lengths to prevent the sig_send function that sends signals from being interrupted. In the case of a process sending a signal to another process, we place a mutex around sig_send such that sig_send will not be interrupted until it has completely finished sending the signal.

In the case of a process sending itself a signal, we use a separate semaphore/event pair instead of the mutex. sig_send starts by resetting the event and incrementing the semaphore that flags the signal handler to process the signal. After the signal is processed, the signal handler signals the event that it is done. This process keeps intraprocess signals synchronous, as required by POSIX.

Most standard UNIX signals are provided. Job control works as expected in shells that support it.

5.8. Sockets

Socket-related calls in Cygwin32 simply call the functions by the same name in Winsock, Microsoft's implementation of Berkeley sockets. Only a few changes were needed to match the expected UNIX semantics — one of the most troublesome differences was that Winsock must be initialized before the first socket function is called. As a result, Cygwin32 has to perform this initialization when appropriate. In order to support sockets across fork calls, child processes initialize Winsock if any inherited file descriptor is a socket.

Unfortunately, implicitly loading DLLs at process startup is usually a slow affair. Because many processes do not use sockets, Cygwin32 explicitly loads the Winsock DLL the first time it calls the Winsock initialization routine. This single change sped up GNU configure times by thirty percent.

5.9. Select

The UNIX select function is another call that does not map cleanly on top of the Win32 API. Much to our dismay, we discovered that the Win32 select in Winsock only worked on socket handles. Our implementation allows select to function normally when given different types of file descriptors (sockets, pipes, handles, and a custom /dev/windows windows messages pseudo-device).

Upon entry into the select function, the first operation is to sort the file descriptors into the different types. There are then two cases to consider. The simple case is when at least one file descriptor is a type that is always known to be ready (such as a disk file). In that case, select returns immediately as soon as it has polled each of the other types to see if they are ready. The more complex case involves waiting for socket or pipe file descriptors to be ready. This is accomplished by the main thread suspending itself, after starting one thread for each type of file descriptor present. Each thread polls the file descriptors of its respective type with the appropriate Win32 API call. As soon as a thread identifies a ready descriptor, that thread signals the main thread to wake up. This case is now the same as the first one since we know at least one descriptor is ready. So `select` returns, after polling all of the file descriptors one last time.

6. Performance

Early on in the development process, correctness was almost the entire emphasis. As Cygwin32 became more complete, performance became a much important issue. We knew that the tools ran much more slowly under Win32 than under Linux on the same machine, but it was not clear at all whether to attribute this to differences in the operating systems or to inefficiencies in Cygwin32.

The lack of a working profiler has made analyzing Cygwin32's performance particularly difficult. Although the latest version of the library includes "real" itimer support, we have not yet found a way to implement virtual itimers. This is the most reliable way of obtaining profiling data since concurrently running processes aren't likely to skew the results. We will soon have a combination of the gcc compiler and the GNU profile analysis tool gprof working with "real" itimer support which will help a great deal in optimizing Cygwin32.

Even without a profiler, we knew of several areas inside Cygwin32 that definitely needed a fresh approach. While we rewrote those sections of code, we used the speed of configuring the tools under Win32 as the primary performance measurement. This choice made sense because we knew process creation speed was especially poor, something that the GNU configure process stresses.

These performance adjustments made it possible to completely configure the development tools under NT with Cygwin32 in only ten minutes and complete the build in just under an hour on a dual Pentium Pro 200 system with 128 MB of RAM. This is reasonably competitive with the time taken to complete this task under a typical flavor of the UNIX operating system running on an identical machine.

7. Ported Software

In addition to being able to configure and build most GNU software, several other significant packages have been successfully ported to the Win32 host using the Cygwin32 library. Following is a list of some of the more interesting ones (most are not included in the free Internet distributions):

- X11R6 client libraries, enabling porting many X programs to the existing free Win32 X servers. Examples of successfully ported X applications include xterm, ghostview, xfig, and xconq.
- xemacs and vim editors.
- GNU inetutils. It is possible to run the inetd daemon as a Windows NT service to enable UNIX-style networking, using a custom NT login binary to allow remote logins with full user authentication. One can achieve similar results under Windows 95/98 by running inetd out of the `autoexec.bat` file, providing a custom 95/98-tailored login binary.
- KerbNet, Cygnus' implementation of the kerberos security system.
- CVS (Concurrent Versions System), a popular version control program based on RCS. Cygnus uses a kerberos-enabled version of CVS to grant secure access to our source code to local and remote engineers.
- ncurses, a library that can be used to build a functioning version of the pager "less".
- ssh (secure shell) client and server.
- PERL 5 scripting language.
- The bash, tcsh, ash, and zsh shells. Full job control is available in shells that support it.
- Apache web server (some source-level changes were necessary).
- TCL/TK 8; also tix, itcl, and expect. (TCL/TK needed non-trivial configuration changes).

Typically, the only necessary source code modification involves specifying binary mode to `open` calls as appropriate. Because our Win32 compiler always generates executables that end in the standard `.exe` suffix, it is also often necessary to make minor modifications to makefiles so that make will expect the newly built executables to end with the suffix.

8. Future Work

Standards conformance is becoming a more important focus. In the last release, all POSIX.1/90 calls are implemented except for `mkfifo` and `setuid`. X/Open Release 4[4] conformance may be a desirable goal, but we have not pursued this yet. While the current version of the library passes most of the NIST POSIX test suite[5] with flying colors, it performs poorly with respect to mimicking the UNIX security model, so there is still room for improvement. When we consider how to implement the `setuid` functionality, we will also look into a secure alternative to the library's usage of the shared memory areas.

Cygwin32 does not yet support applications that use multiple Windows threads, even though the library itself is multi-threaded. We expect to address this shortcoming through the use of locks at strategic points in the DLL. It would also be desirable to implement support for POSIX threads.

Although Cygwin32 allows the GNU development tools that depend heavily on UNIX semantics to successfully run on Win32 hosts, it is not always desirable to use it. A program using a perfect implementation of the library would still incur a noticeable amount of overhead. As a result, an important future direction involves modifying the compiler so that it can optionally link against the Microsoft DLLs that ship with both Win32 operating systems, instead of Cygwin32. This will give developers the ability to choose whether or not to use Cygwin32 on a per-program basis.

9. Proprietary Alternatives

When we started developing Cygwin32, alternatives to writing our own library either did not exist or were not mature enough for our purposes. Today, we know of three proprietary alternatives to Cygwin32: UWIN from AT&T Laboratories, NuTCracker from DataFocus, and OpenNT from Softway Systems.

UWIN[6] ("UNIX for Windows") was developed by David Korn for AT&T Laboratories. Its architecture and API appears to be quite similar to our library. Its single biggest advantage over Cygwin32 is probably its more complete support for the UNIX security model. UWIN binaries are available for free non-commercial use, but its source code is not available.

NuTCracker, by DataFocus, is another proprietary product that is built on top of the Win32 subsystem. Version 4.0 of the product appears to be quite complete, including such features as support for POSIX threads.

OpenNT from Softway Systems[7] takes a markedly different approach by providing a capable POSIX subsystem for Windows NT, implemented with the Windows NT source code close at hand. At least in principle, writing a separate POSIX subsystem should result in better performance because of the lack of overhead imposed when implementing a library on top of the Win32 subsystem. More importantly, by avoiding the compromises inherent in supporting both Win32 and POSIX calls in one application, it should be possible for OpenNT to conform more strictly to the relevant standards.

However, there are two substantial drawbacks to OpenNT's approach. The first is that it is not possible to mix UNIX and Win32 API calls in one application, a feature that is highly desirable if you are attempting to do a full native Win32 port of a UNIX program gradually, one module at a time. The second drawback is that OpenNT does not and cannot support Windows 95/98, a requirement for many applications, including the GNUPro development tools.

The lack of source code, coupled with the licensing fees associated with each of these commercial offerings, might still have required us to have written our own library if we were faced with the same porting challenge today.

10. Summary and Conclusions

Cygwin32 is a UNIX-compatibility library that can be used to port UNIX software to Win32 operating systems. In this paper, I have examined our motivations for writing Cygwin32. I have analyzed its architecture in some detail, paying extra attention to those areas where UNIX and Win32 differ the most. I have listed examples of successfully ported software and touched on performance issues. I have discussed where we expect to take Cygwin32 in the future. Finally, I have described the proprietary alternatives to our library.

As you can see from the list of ported software presented earlier in this paper, Cygwin32 can be used to facilitate greatly the process of porting significant UNIX applications to Win32 hosts. For some applications, it may be desirable to invest in a true native Win32 port in order to remove the overhead imposed by Cygwin32. However, the increased portability and time saved by using Cygwin32 should make it an attractive option in many situations.

11. Availability

Please consult our project WWW page to obtain more information about Cygwin32, including how to download the latest source code and binary release:

http://www.cygnus.com/misc/gnu-win32

For more information about the GNUPro development tools, please visit:

http://www.cygnus.com/product/gnupro.html

12. Acknowledgements

The author wishes to thank the many other people who have helped write Cygwin32, in particular Steve Chamberlain who wrote the original implementation of the library. Jeremy Allison, Doug Evans, Christopher Faylor, Philippe Giacinti, Tim Newsham, Sergey Okhapkin, and Ian Taylor have all made significant contributions to the library. The author also appreciates the feedback and proofreading help given to him by Eric Bachalo, Chip Chapin, Christopher Faylor, Kathleen Jones, Robert Richardson, Stan Shebs, Sonya Smallets, and Ethan Solomita, as well as from Stephan Walli, his USENIX paper advisor.

13. Trademarks

GNUPro is a registered trademark of Cygnus Solutions. Windows NT, Windows 95, Windows 98, Win32, Windows, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. UNIX is a trademark of the Open Group. OpenNT is a trademark of Softway Systems. All other trademarks belong to their respective holders.

[1] D.J. Delorie: The DJGPP Project. Available from http://www.delorie.com/djgpp.

[2] ISO/IEC 9945-1:1996. (ANSI/IEEE Std 1003.1, 1996 Edition) — POSIX Part 1: System Application Program Interface (API) [C Language].

[3] ISO/IEC 9899:1990, Programming Languages — C.

[4] The X/Open Release 4 CAE Specification, System Interfaces and Headers, Issue 4, Vol. 2, X/Open Co, Ltd., 1994.

[5] NIST POSIX test suite. Available from http://www.itl.nist.gov/div897/ctg/posix_form.htm.

[6] Korn, David G. UWIN — UNIX for Windows. Proceedings of the 1997 USENIX Windows NT Annual Technical Conference.

[7] Walli, Stephen R. OpenNT: UNIX Application Portability to Windows NT via an Alternative Environment Subsystem. Proceedings of the 1997 USENIX Windows NT Workshop Proceedings.

Win32 API Emulation on UNIX for Software DSM

Sven M. Paas, Thomas Bemmerl, Karsten Scholtyssik
RWTH Aachen, Lehrstuhl für Betriebssysteme
Kopernikusstr. 16, D-52056 Aachen, Germany
e-mail: `contact@lfbs.rwth-aachen.de`

Abstract. *This paper presents a new Win32 API emulation layer called nt2unix. It supports source code compatible Win32 console applications on UNIX. We focus on the emulation of specific Win32 features used for systems programming like exception handling, virtual memory management, Windows NT multithreading / synchronization and the WinSock API for networking. As a case study, we ported the all-software distributed shared memory (DSM) system SVMlib - consisting of about 15.000 lines of C++ code written natively for the Win32 API - to Sun Solaris 2.5 with absolutely no source code changes.*

1 Introduction

While there exist numerous products and toolkits designed for porting software from UNIX[1] to Windows NT on source code level (like OpenNT from Softway Systems, Inc. [14], or other tools [17]), much less effort has been conducted to provide a toolkit to port Windows NT applications, especially those using the Win32 API directly, to UNIX. Apart from some very expensive commercial products in this area like *Wind/U* from Bristol Technology, Inc. [2] or *MainWin XDE* from MainSoft Corp. [9], a small low cost solution is not known to the authors. With the emerging importance of Windows NT [10], more and more applications are newly developed for the Win32 API, so a migration path to support the various UNIX flavors even for low level applications is highly desirable. In this paper, we propose a library based approach to achieve source code level compatibility for a specific subset of the Win32 API under the UNIX operating system.

2 The nt2unix Emulation Layer

In this section, we introduce the functionality of nt2unix and its strategy to implement a relevant subset of the Win32 API on Solaris [15], a popular UNIX System V implementation. Since a complete implementation of the Win32 API under UNIX is not practicable, we had to decide which features to support. As our focus lies on systems programming, we chose the following function groups to form a reasonable subset:

- *Windows NT Multithreading and Synchronization.* This group includes functions for creating, destroying, suspending and resuming preemptive threads. It also includes functions to synchronize concurrent threads and TLS (thread local storage) functions.
- *Virtual Memory Management.* This group includes the interface to the virtual memory (VM) manager as well as functions for memory mapped I/O.
- *Windows NT Exception Handling.* Win32 supports user level handlers to catch special exceptions as well as certain error handling routines. These functions form another group to be supported.
- *Networking.* This group concerns networking, which includes the complete WinSock API (restricted to the TCP/IP protocol family, however).

Naturally, an emulation layer firstly has to support the basic data types found in the various C++ header files of the Win32 API. nt2unix supports most specific simple Win32 data types, like `DWORD`, `BOOL`, `BYTE` and so on. The much more interesting problems arise from the implementation of certain functions. Some specific problems we encountered are presented in the next sections.

2.1 Multithreading and Synchronization

In order to support Windows NT multithreading, nt2unix must keep track of thread associated data normally the Windows NT kernel stores. This data includes:

- The *state* of a thread (running, suspended or terminated) - by default, a thread is created in running state;
- The *suspend counter* of a thread (a concept unknown in the Solaris or POSIX thread API);
- The *exit code* of the thread.

nt2unix uses the *Standard Template Library* (STL) type `map` to store the above information for each thread. The entries in the map are indexed by the Windows NT handle of the thread. Accesses to this map are protected by a special lock object of class `CriticalSection`, which is a comfortable wrapper around the Windows NT `CRITICAL_SECTION` type:

```
class CriticalSection {
  public:
    CriticalSection::CriticalSection() {
      InitializeCriticalSection(&cs);
    }
    CriticalSection::~CriticalSection() {
      DeleteCriticalSection(&cs);
    }
    inline void CriticalSection::enter() {
      EnterCriticalSection(&cs);
    }
    inline void CriticalSection::leave() {
      LeaveCriticalSection(&cs);
    }
  protected:
    CRITICAL_SECTION cs;
};

struct ThreadInfo {
  ThreadInfo::ThreadInfo() {
    ThreadInfo::init(THREAD_RUNNING);
  }
  ThreadInfo::ThreadInfo(DWORD aState) {
    ThreadInfo::init(aState);
  }
  inline void ThreadInfo::init(DWORD aState) {
    state = aState;
    suspendCount = 0;
    exitCode = 0;
    threadHasBeenResumed = FALSE; // see below
  }
  DWORD suspendCount;
  DWORD state;
  DWORD exitCode;
  volatile BOOL  threadHasBeenResumed;
  // A special flag to synchronize
  // SuspendThread() / ResumeThread()
 };

typedef map<HANDLE, ThreadInfo,
        less<HANDLE> > ThreadInfoMap;
static ThreadInfoMap ThreadInfos;
static CriticalSection ThreadInfoLock;
```

Important problems occur in order to support the Win32 functions **SuspendThread**() and **ResumeThread**(). At first glance, it seems obvious that these two functions can easily be emulated by the Solaris functions **thr_suspend**() and **thr_resume**(). However, this is not the case, since there is a lost signal problem to be avoided when a thread suspends. The situation using the POSIX thread API is even worse, because there are no functions available for resuming or suspending threads anyway.

To understand this, we have a deeper look at our implementation of **SuspendThread**() using the Solaris thread API. When this function is called, the lock protecting the thread data is acquired. Afterwards, the suspend counter of the thread is incremented, if possible. If the old suspend counter is zero, two cases may occur: the thread may suspend itself or another thread. If the first case is true, the lock is released before actually calling **thr_suspend**() to avoid deadlock. In the second case, a lost signal problem must be avoided, since under Solaris, resuming threads doesnot work in advance, that is, resume actions are not queued if the target thread is not yet suspended at all. Our solution to this problem is to let the **ResumeThread**() implementation poll until the thread which has to be resumed has indicated its new state by setting a special flag, threadHasBeenResumed. So the code for **SuspendThread**() looks like the following:

```
DWORD SuspendThread(HANDLE hThread) {
  BOOL same = FALSE;
  // this flag indicates whether
  // a thread suspends itself.
  // If same == TRUE, we must avoid a
  // "lost signal" problem, see below.
  ThreadInfoLock.enter();
  ThreadInfoMap::iterator thisThreadInfo =
    ThreadInfos.find(hThread);
  if (thisThreadInfo != ThreadInfos.end()) {
    // found it.
    DWORD oldSuspendCount =
      (*thisThreadInfo).second.suspendCount;
    if (oldSuspendCount < MAXIMUM_SUSPEND_COUNT)
      (*thisThreadInfo).second.suspendCount++;
    if (oldSuspendCount < 1) {
      (*thisThreadInfo).second.state =
        THREAD_SUSPENDED;
      if (same =
        (thr_self() == (thread_t)hThread)){
        // if the thread suspends itself,
        // we must release the lock.
        (*thisThreadInfo).second.\
          threadHasBeenResumed = FALSE;
        ThreadInfoLock.leave();
      }
      // DANGER!!! If at this point, another
      // thread is scheduled in ResumeThread(),
      // the resume „signal" may get lost.
      // To avoid this, ResumeThread()
      // polls until the thread is really
      // resumed, i.e. until
      // threadHasBeenResumed == TRUE.
      if (thr_suspend((thread_t)hThread)) {
        perror("thr_suspend()");
        return 0xFFFFFFFF;
      }
      (*thisThreadInfo).second.\
        threadHasBeenResumed = TRUE;
      if (!same)
        ThreadInfoLock.leave();
    } else
      // thread is already sleeping
      ThreadInfoLock.leave();
    return oldSuspendCount;
  }
  // Thread not found.
  ThreadInfoLock.leave();
  return 0xFFFFFFFF;
}
```

The corresponding **ResumeThread**() code is as follows:

```
DWORD ResumeThread(HANDLE hThread) {
  ThreadInfoLock.enter();
  ThreadInfoMap::iterator thisThreadInfo =
    ThreadInfos.find(hThread);
  if (thisThreadInfo != ThreadInfos.end()) {
    // found it.
    DWORD oldSuspendCount =
      (*thisThreadInfo).second.suspendCount;
    if (oldSuspendCount > 0) {
      (*thisThreadInfo).second.suspendCount--;
      if (oldSuspendCount < 2) {
        // oldSuspendCount == 1 -> new
        // value is 0 -> really resume thread
        (*thisThreadInfo).second.state =
          THREAD_RUNNING;
        do { // Loop until the target thread
             // is really resumed.
          if (thr_continue((thread_t)hThread)){
            ThreadInfoLock.leave();
            return 0xFFFFFFFF;
          }
          // Give up the CPU so that the resumed
          // thread has a chance to update the
          // associated threadHasBeenResumed
          // flag.
          thr_yield();
        } while (!(*thisThreadInfo).\
          second.threadHasBeenResumed);
      }
    }
    ThreadInfoLock.leave();
    return oldSuspendCount;
  }
  // thread not found.
  ThreadInfoLock.leave();
  return 0xFFFFFFFF;
}
```

Other caveats occur in order to support synchronization function like **EnterCriticalSection**() and **LeaveCriticalSection**(), because under Windows NT the CRITICAL_SECTION objects can be acquired recursively (that is, a lock owning thread may acquire the same lock without deadlocking), while Solaris / POSIX thread mutexes are not. The solution to this problem is again to reinvent the wheel and try to emulate this behavior. We use the standard Windows NT type

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;
    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;
    HANDLE LockSemaphore;
    DWORD Reserved;
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

for each critical section object and its thus possible to keep track of recursive lock acquires and releases:

```
WINBASEAPI VOID WINAPI EnterCriticalSection(
  LPCRITICAL_SECTION lpCriticalSection) {
  thread_t me = thr_self();
  if (lpCriticalSection->OwningThread ==
    (HANDLE)me) {
    // I have the lock.
    // This cannot be a race condition.
    lpCriticalSection->RecursionCount++;
    return;
  }
  if(mutex_lock((mutex_t *)(lpCriticalSection
    ->LockSemaphore))) {
    DBG("mutex_lock() failed"); return;
  }
  // got it. I must be the first thread:
  if (lpCriticalSection->RecursionCount) {
    DBG("RecursionCount != 0"); return;
  }
  lpCriticalSection->RecursionCount = 1;
  lpCriticalSection->OwningThread = (HANDLE)me;
  return;
}
```

If the thread acquiring a lock is the same thread already owning the lock, the recursion counter is incremented. This cannot be a race condition, because no other thread can change this value at the time the lock is blocked. Otherwise, **mutex_lock**() (or **pthread_mutex_lock**() in the POSIX version) is called. If the lock is successfully acquired, the recursion counter must be 0 and is set to 1 afterwards.

```
WINBASEAPI VOID WINAPI LeaveCriticalSection(
    LPCRITICAL_SECTION lpCriticalSection) {
    thread_t me = thr_self();
    if (lpCriticalSection->OwningThread ==
      (HANDLE)me) {
      lpCriticalSection->RecursionCount--;
      if(lpCriticalSection->RecursionCount < 1) {
        lpCriticalSection->OwningThread =
          (HANDLE)0xFFFFFFFF;
        if(mutex_unlock((mutex_t *)
          lpCriticalSection->LockSemaphore))
          DBG("mutex_unlock() failed");
      }
    } else
      DBG("not lock owner");
  return;
}
```

When leaving a critical section, the recursion counter is decremented if the call was recursive. If the counter is 0 no thread owns the lock anymore, hence **mutex_unlock**() (or **pthread_mutex_unlock**() in the POSIX version) must be called. It is an error if the caller of **LeaveCriticalSection**() was not the owner of the lock.

2.2 Virtual Memory Management

Win32 supports an interface to the VM system, especially to protect and map virtual memory pages. Like for threads, nt2unix has to keep track of data for each file mapping in the system. nt2unix stores the following information for each mapping:

```
  struct FileMapping {
    LPVOID lpBaseAddress;
    // the virtual base address of the mapping
  DWORD dwNumberOfBytesToMap;
    // the mapping size in bytes
  HANDLE hFileMappingObject;
    // the file handle
  char FileName[MAX_PATH];
    // the file name
  DWORD refcnt;
    // the number of references to the mapping
};
static vector<FileMapping> FileMappings;
```

The virtual base address for the mapping is stored in `lpBaseAddress`, while the size of the mapping object in bytes is stored in `dwNumberOfBytes`. The handle of the mapped file and / or its file name are stored in `hFileMappingObject` and `FileName[]`, respectively. The above `struct` is allocated for a specific mapping by calling our emulation of **`CreateFileMapping`**() or **`CreateFileMappingA`**(), respectively. It is deallocated if the `refcnt` counter keeping track of the open references to the mapping for the mapping is 0. Using a STL-style `vector` of mappings, Windows NT mapping is achieved by using **`mmap`**():

```
WINBASEAPI LPVOID WINAPI MapViewOfFileEx(
    HANDLE hFileMappingObject,
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    DWORD dwNumberOfBytesToMap,
    LPVOID lpBaseAddress ) {
    int prot = 0, flags = 0; LPVOID ret;
  if (dwFileOffsetHigh > 0)
    DBG("Ignoring dwFileOffsetHigh");
  // Filter the protection bits
  // and mapping flags ...
  prot = dwDesiredAccess & FILE_MAP_ALL_ACCESS;
  flags = ((dwDesiredAccess & FILE_MAP_COPY) ==
    FILE_MAP_COPY) ? MAP_PRIVATE : MAP_SHARED;
  if (lpBaseAddress)
    flags |= MAP_FIXED;
  // Search and update the mapping
  // in the vector.
  vector<FileMapping>::iterator i =
    FileMappings.begin();
  while(i != FileMappings.end() &&
        i->hFileMappingObject !=
        hFileMappingObject)
    i++;
  if (i != FileMappings.end()) {
    if (dwNumberOfBytesToMap)
      i->dwNumberOfBytesToMap =
        dwNumberOfBytesToMap;
  } else
    return 0;
  if ((ret =
      (LPVOID)mmap((caddr_t)lpBaseAddress,
      (size_t)i->dwNumberOfBytesToMap, prot,
      flags, (int)hFileMappingObject,
      (off_t)dwFileOffsetLow)) ==
      (LPVOID)MAP_FAILED)
    return 0;
  if (mprotect((caddr_t)ret,
      (size_t)i->dwNumberOfBytesToMap,
      prot) == -1)
    perror("mprotect()");
  return ret;
}
```

The similar function **`MapViewOfFile`**() is now easily implemented by calling **`MapViewOfFileEx`**() with the last parameter *lpBaseAddress* set to 0. The mentioned `refcnt` for each mapping is decremented by a call to **`UnMapViewOfFile`**(). However, the above implementation has a 4 GB limit concerning the maximum size of the mapped file, because this is the maximum file mapping size possible under Solaris.

The memory access protection bits of a file mapping under Windows NT have more a less equivalent values under UNIX. However, not all bit masks are supported, namely `PAGE_GUARD` and `PAGE_NOCACHE`:

Windows NT Protection Bits	UNIX Bits
PAGE_READONLY	PROT_READ
PAGE_READWRITE	(PROT_READ | PROT_WRITE)
PAGE_NOACCESS	PROT_NONE
PAGE_EXECUTE	PROT_EXEC
PAGE_EXECUTE_READ	(PROT_EXEC |PROT_READ)
PAGE_EXECUTE_READWRITE	(PROT_EXEC | PROT_READ | PROT_WRITE)
PAGE_GUARD	n/a
PAGE_NOCACHE	n/a

2.3 Windows NT Exception Handling

Windows NT provides two means of delivering exceptions to user level processes:

- by embracing the code with a `__try{} ... __except(){}` block;
- by installing an exception handler calling **`SetUnhandledExceptionFilter`**().

Because the first method is a proprietary language extension, only the second method is supported by nt2unix. Exceptions are mapped to semantically more or less equivalent UNIX-style signals, like denoted in the following table. Note that not all exception codes of Windows NT have meaningful counterparts in a UNIX environment:

Windows NT **`EXCEPTION_*`** Code	UNIX Signal
ACCESS_VIOLATION	SIGSEGV
FLT_INVALID_OPERATION	SIGFPE

Windows NT **EXCEPTION_*** Code	UNIX Signal
ILLEGAL_INSTRUCTION	SIGILL
IN_PAGE_ERROR	SIGBUS
SINGLE_STEP	SIGTRAP

The emulation of Windows NTs exception handling requires carefully converting UNIX-style types like siginfo_t and ucontext_t to a Windows NT-style struct EXCEPTION_POINTERS. To fill in this struct, the stack frame of the signal handler method must be examined, which is very system dependent.

For example, the page fault info is extracted in a SIGSEGV handler for Solaris SPARC (__SPARC), Solaris x86 (__X86) and Linux x86 (__LINUXX86) in the following way:

```
switch (sig) {
case SIGSEGV:
    // A segmentation violation.
    ExceptionInfo.ExceptionRecord->
      ExceptionCode = EXCEPTION_ACCESS_VIOLATION;
    ExceptionInfo.ExceptionRecord->
      ExceptionInformation[0] =
#if defined(__SPARC)
        (*(unsigned *)((ucontext_t*)uap)
          ->uc_mcontext.gregs[REG_PC] & (1<<21));
#elif defined(__X86)
        (((ucontext_t*)uap)->
          uc_mcontext.gregs[ERR] & 2);
#elif defined(__LINUXX86)
        stack[14] & 2;
#endif
    if (ExceptionInfo.ExceptionRecord->
      ExceptionInformation[0])
      ExceptionInfo.ExceptionRecord->
        ExceptionInformation[0] = 1;
        // 1 == write access
    ExceptionInfo.ExceptionRecord->
      ExceptionInformation[1] =
#ifdef __LINUXX86
        stack[22];
#else
        (DWORD)sip->si_addr;
#endif
    break;

    // other signals processed here ...
}
```

If a SIGSEGV is caught, the exception type is set to EXCEPTION_ACCESS_VIOLATION. In the next statements, the type of the fault (read or write) as well as the faulting address must be extracted from the stack. Under Solaris SPARC, the type of the fault is coded in bit 21 of the REG_PC register, while under Solaris x86, bit 2 of the ERR register contains this information. Under Linux x86, bit 2 of the stack at position 14 stores this bit of ERR according to the Linux 2.0 kernel source.

The faulting address under Solaris is located under sip->si_addr, where sip of type siginfo_t* is the second parameter of the signal handler function installed. Under Linux, the value is found at position 22 of the signal handler stack.

Of course, this code is not portable and must be implemented again for each UNIX derivative.

2.4 Networking

The standard protocol family available under UNIX is TCP/IP. With nt2unix, we map the WinSock API with respect to this protocol to the standard BSD sockets API. The main difference between the Windows NT and the BSD socket API is due to some new Windows NT data types and definitions:

```
typedef int                SOCKET;
#define INVALID_SOCKET     (SOCKET)(-1)
#define SOCKET_ERROR       (-1)
```

Additionally, Windows NT defines the Windows Sockets (WinSock) API error codes (WSA*). The only real problem while emulating the WinSock API under BSD was found for the **select**() call. This function has the prototype

```
int select(int nfds, fd_set FAR *readfds,
           fd_set FAR *writefds,
           fd_set FAR *exceptfds,
           const struct timeval FAR * timeout);
```

A source of hard to find programming errors is that the fd_set data type is usually implemented as a bit mask under UNIX, while Windows NT implements this data type as an ordinary array. That is the reason why Windows NT ignores the first parameter nfds which defines the highest bit to be scanned while waiting for pending input. That is, you can unfortunately write Windows NT code using **select**() which doesnot run under BSD.

2.5 Summary

The following table shows a summary of all functions implemented within nt2unix.

	Win32 Functions emulated	Emulation is based on
Multi-threading	CreateThread()	thr_create()
	GetCurrentThread()	thr_self()
	GetCurrentThreadId()	thr_self()
	ExitThread()	thr_exit()
	TerminateThread()	thr_kill()
	GetExitCodeThread()	STL
	SuspendThread()	thr_self(), thr_suspend()
	ResumeThread()	thr_yield(), thr_resume()
	Sleep()	thr_yield(), thr_suspend(), poll()

	Win32 Functions emulated	Emulation is based on
Thread Synchronization	InitializeCriticalSection()	mutex_init()
	DeleteCriticalSection()	mutex_destroy()
	EnterCriticalSection()	mutex_lock()
	LeaveCriticalSection()	mutex_unlock()
Thread Local Storage (TLS)	TlsAlloc()	thr_keycreate()
	TlsGetValue()	thr_getspecific()
	TlsSetValue()	thr_setspecific()
	TlsFree()	pthread_key_delete()
Object Handles	CloseHandle()	close()
	DuplicateHandle()	dup(), dup2()
	WaitForSingleObject()	thr_join()
Process Functions	GetCurrentProcess()	getpid()
	GetCurrentProcessId()	getpid()
	ExitProcess()	exit()

VM Management	VirtualAlloc()
	VirtualFree()
	VirtualProtect()
	VirtualLock()
	VirtualUnlock()
		mmap(), valloc(),
		mprotect()
		mprotect(), free()
		mprotect(),
		memcntl()
		mlock()
		munlock()

Memory Mapped I/O	MapViewOfFile()	mmap()
	MapViewOfFileEx()	mmap()
	UnmapViewOfFile()	munmap()
	CreateFileMapping()	STL
Error Handling	WSAGetLastError()	errno
	GetLastError()	errno
	SetLastError()	errno
	WSASetLastError()	errno
WinSock API	WSAStartup()	-
	WSACleanup()	-
	closesocket()	close()
	ioctlsocket()	ioctl()
	all BSD-style functions!	socket(5) family
Exception Handling	SetUnhandledExceptionFilter()	sigaction()
	GetExceptionInformation()	
	UnhandledExceptionFilter()	
Miscellaneous	GetSystemInfo()	sysinfo()
	GetComputerName()	gethostname()
	QueryPerformanceFrequency()	-
	QueryPerformanceCounter()	gettimeofday()

3 A Case Study: SVMlib

3.1 Overview

SVMlib [11, 16] (*Shared Virtual Memory Library*) is an all-software, page based, user level shared virtual memory [1] subsystem for clusters of Windows NT workstations. It is one of the first (among [7] and [12]) SVM systems for this operating system. The source code of SVMlib consists of about 15.000 lines of C++ code written natively for the Win32 API. The library has been designed to benefit from several Windows NT features like preemptive multithreading and support for SMP machines. Unlike most software DSM systems, SVMlib itself is truly multithreaded. It also allows users to create several preemptive user threads to speed up the computation on SMP nodes in the cluster. Currently the library uses TCP/IP sockets for communication purposes but it will also support efficient message passing using the Dolphin implementation [3] of the *Scalable Coherent Interface* (SCI) [5].

SVMlib provides a C/C++ API that allows the user to create and destroy regions of virtual shared memory that can be accessed fully transparently. Different synchronization primitives such as barriers and mutexes are part of the API. To keep track of accesses to the shared regions, SVMlib handles page faults within the regions via structured exception handling provided by the C++ run time system of Windows NT.

At the current stage, two different memory consistency models are supported by three different consistency protocols. The first consistency model offers the widely used though fairly inefficient *sequential consistency* [8] model. This model is supported by single writer as well as multiple writer protocols. Secondly, the distributed lock based *scope consistency* [6] is implemented.

3.2 Design Issues

When designing an SVM system, several design choices have to be made. When we started this project our primary goal was to develop a highly flexible and extendable research instrument. We therefore decided to build SVMlib as a set of independent modules where each can be exchanged without influencing the other modules.

Another important choice was the platform to build SVMlib on. As Windows NT is a modern operating system with some interesting features like true preemptive kernel threads, SMP support and a rich API we decided to use workstations running Windows NT as the primary platform. Figure 1 shows the overall design of SVMlib. On the top level four modules are used.

The first is the *memory manager* that handles the creation and destruction of shared memory regions, catches page faults and implements the memory dependent part of the user interface. The memory manager manages a set of regions where each region can use a different consistency model and coherence protocol.

The second part is the *lock manager* that provides an interface that allows the user to create and destroy primitives for distributed process synchronization - mutexes as well

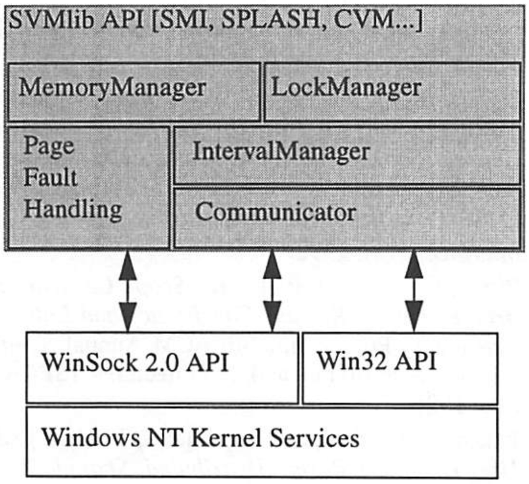


Figure 1: SVMlib layers

as global barriers and semaphores.

For internode communication purposes the *communicator* is used. The user will never directly use this module. It is for internal purposes only. The communicator provides a simple interface containing a barrier, a broadcast algorithm and the possibility to send messages to each other node. This module has been designed to be active itself. To take advantage of the SMP support of Windows NT the communicator uses threads to handle incoming messages.

The last main module is the *interval manager* that allows to implement weak consistency models like lazy release consistency or the currently used scope consistency. The user will never have to access this module directly. It is used as a bridge between the memory and the lock manager when weak consistency models are used. This is needed because both locks and memory pages handle a part of the weak consistency model.

SVMlib provides several API personalities to the application programmer. First of all, a native C and C++ API is provided. For compatibility to other SVM systems and existing shared memory implementations, other interfaces to shared memory programming are supported. Currently, these interfaces include the *Shared Memory Interface* (SMI) [4], the macro interface of *Stanford Parallel Applications for Shared Memory* (SPLASH) [18] and the *Coherent Virtual Machine* (CVM) [13]. Other interfaces are planned to be supported in the future.

3.3 Performance Impact of the Emulation

Using nt2unix, we ported the source code of SVMlib to Sun Solaris 2.5.1 with *absolutely* no source code changes. Though the development of nt2unix was in fact driven by our goal to port SVMlib to UNIX, this was very surprising, since, at first glance, a DSM implementation naturally is very system dependent. To show the impact of the Win32 emulation, we give usual metrics characterizing the performance of the SVM library:

- *Page Fault Detection Time.* This value includes the mean time from the occurrence of a processor page fault on a protected page to the entrance of the handling routine. That is, this time includes all operating system overhead to deliver a page fault exception to user code. Note that there seems to be no difference between the Windows NT Server and NT Workstation (WS) version with respect to exception handling. We compared these values with user level page fault detection under Solaris 2.5.1 for Intel and SPARC using nt2unix, respectively. As mentioned, under UNIX, the memory exception handling mechanism of Windows NT is emulated by catching the `SIGSEGV` signal.

	SuperSPARC, 50 MHz	Pentium, 133 MHz	Pentium Pro, 200 MHz
Windows NT 4.0 Server / WS	-	28 µs	19 µs
Solaris 2.5.1 & nt2unix	135 µs	92 µs	48 µs

- *Page Fault Time.* This value includes the mean time to *handle* one page fault. This time excludes the page fault detection time mentioned above. It includes the overhead due to the coherence protocol and communication subsystem. In the current implementation, the times measured are mainly influenced by the high TCP/IP latency. The measurements were made using the FFT application of the set of CVM [13] examples. This application implements a Fast Fourier Transformation on a 64 x 64 x 16 array. The coherence protocol used is a multiple reader / single writer protocol implementing sequential consistency. We compared three configurations running FFT: (1) *CVM on Solaris*: the CVM system running on Solaris 2.5.1, Sun SS-20, Ethernet; (2) *SVMlib on nt2unix*: the Solaris version of SVMlib, running on the same platform as (1), but with nt2unix emulation layer; (3) *SVMlib on Win32*: the native Win32 version of SVMlib, running on Windows NT 4.0, Intel Pentium-133, FastEthernet. Naturally, the Win32 time values mainly reflect the improved network performance of FastEthernet.

Nodes	**Read / Write / Average Fault Time [ms] (CVM on Solaris)**	**Read / Write / Average Fault Time [ms] (SVMlib on nt2unix)**	**Read / Write / Average Fault Time [ms] (SVMlib on Win32)**
2	11.3 / 0.8 / 4.4	4.5 / 1.3 / 2.2	3.4 / 1.1 / 1.8

Nodes	Read / Write / Average Fault Time [ms] (CVM on Solaris)	Read / Write / Average Fault Time [ms] (SVMlib on nt2unix)	Read / Write / Average Fault Time [ms] (SVMlib on Win32)
3	12.0 / 0.8 / 5.8	4.6 / 1.8 / 2.7	3.4 / 1.4 / 2.3
4	16.7 / 0.9 / 7.1	4.9 / 1.8 / 3.1	4.0 / 1.5 / 2.4

It is clear that the above measurements are not sufficient to determine the overall performance of nt2unix, but they show reasonable results with respect of key functions used to implement SVMlib on Windows NT: multithreading, networking and exception handling.

3.4 Summary and Conclusion

In this paper, we introduced nt2unix, a library providing an important subset of the Win32 API on UNIX based systems. The library makes it possible to port Win32 console applications to UNIX with much less effort. The first version of nt2unix was developed and tested on Solaris 2.5 for SPARC and Intel Processors, respectively. At the moment, we are extending the implementation to support more generic UNIX platforms and POSIX interfaces:

- The current version supports the POSIX thread API additionally to the Solaris thread API. This required slightly different implementation of **ResumeThread**() and **SuspendThread**(), because POSIX doesnot include functions equivalent to Solaris **thr_resume**() and **thr_suspend**().
- We have a running Linux/x86 version of nt2unix using a POSIX thread library. We found that especially the exception handling is very system dependent, because the signal handler stack frame has to be inspected to extract the detailed exception information.

As a case study, we ported a complex DSM system with no source code changes at all from Windows NT to Solaris. We found that the performance impact of the emulation is acceptable. The complete source code of the nt2unix library is available upon request, please e-mail to `contact@lfbs.rwth-aachen.de`.

References

[1] Berrendorf, R.; Gerndt, M.; Mairandres, M.; Zeisset, S.: *A Programming Environment for Shared Virtual Memory on the Intel Paragon Supercomputer*, ISUG Conference, Albuquerque, 1995

[2] Bristol Technology Inc., URL: `http://www.bristol.com/`

[3] Dolphin Interconnect Solutions: *PCI-SCI Cluster Adapter Specification.* Jan. 1996.

[4] Dormanns, M.; Sprangers, W.; Ertl, H.; Bemmerl, T.: *A Programming Interface for NUMA Shared-Memory Clusters.* Proc. High Perf. Comp. and Networking (HPCN), pp. 698-707, LNCS 1225, Springer, 1997.

[5] IEEE: *ANSI/IEEE Std. 1596-1992, Scalable Coherent Interface (SCI).* 1992.

[6] Iftode, L.; Singh, J. P.; Li, K.: *Scope Consistency: A Bridge between Release Consistency and Entry Consistency.* In Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96), June 1996

[7] Itzkovitz, A., Schuster, A., Shalev, L.: *Millipede: a User-Level NT-Based Distributed Shared Memory System with Thread Migration and Dynamic Run-Time Optimization of Memory References*, Proc. of the USENIX Windows NT Workshop, Seattle, 1997

[8] Lamport, L.: *How to make a multiprocessor computer that correctly executes multiprocess programs*, IEEE Transactions on Computers, C-28(9), pp. 690-691, September 1979

[9] MainSoft Corp., URL: `http://www.mainsoft.com/`

[10] Microsoft Windows NT Homepage, URL: `http://www.microsoft.com/ntserver/`

[11] Paas, S. M.; Scholtyssik, K.: *Efficient Distributed Synchronization within an all-software DSM system for clustered PCs.* 1st Workshop Cluster-Computing, TU Chemnitz-Zwickau, November 6-7, 1997

[12] Speight, E., Bennett, J. K.: *Brazos: A Third Generation DSM System*, Proc. of the USENIX Windows NT Workshop, Seattle, 1997

[13] Thitikamol, K.; Keleher, P.: *Multi-Threading and Remote Latency in Software DSMs.* In: 17th International Conference on Distributed Computing Systems, May 1997

[14] Softway Systems, Inc. URL: `http://www.softway.com/`

[15] Sunsoft Solaris Homepage, URL: `http://www.sun.com/software/solaris/`

[16] SVMlib Project Homepage, URL: `http://www.lfbs.rwth-aachen.de/~sven/SVMlib/`

[17] UNIX to NT resource center, URL: `http://www.nentug.org/unix-to-nt/`

[18] Woo, S. C.; Moriyoshi Ohara, M.; Torrie, E.; Singh, J. P., and Gupta, A.: *The SPLASH-2 Programs: Characterization and Methodological Considerations.* In Proc. of the 22nd International Symposium on Computer Architecture, pp. 24-36, Santa Margherita Ligure, Italy, June 1995

[1] UNIX is a registered trademark of The Open Group licensed exclusively in conjunction with a brand program.

NT-SwiFT: Software Implemented Fault Tolerance on Windows NT

Yennun Huang
P. Emerald Chung
Chandra Kintala

Bell Laboratories,
Lucent Technologies, Inc.
600 Mountain Avenue
Murray Hill, NJ 07974

Chung-Yih Wang[1]
De-Ron Liang[1]

Institute of Information Science
Academia Sinica
Taipei, Taiwan
R.O.C.

Abstract

More and more high available applications are implemented on Windows NT. However, the current version of Windows NT (NT4) does not provide some facilities that are needed to implement these fault tolerant applications. In this paper, we describe a set of components collectively named NT-SwiFT (Software Implemented Fault Tolerance) which facilitates building fault-tolerant and highly available applications on Windows NT. NT-SwiFT provides components for automatic error detection and recovery, checkpointing, event logging and replay, communication error recovery, incremental data replications, IP packets re-routing, etc. SwiFT components were originally designed on UNIX. The UNIX version was first ported to NT to run on UWIN [Korn97]. Gradually a large portion of the software has been re-implemented to take advantage of native NT system services. This paper describes these components and compares the differences in the UNIX and NT implementations. We also describe some applications using these components and discuss how to leverage NT system services and cope with some missing features.

1. Introduction

Windows NT has become a popular and viable computing platform for critical applications due to its many useful features and low hardware costs. The telecommunication industry has also started to build fault-tolerant and highly available applications on NT. To achieve high reliability and availability in a distributed environment, three types of techniques have been deployed, namely, transaction processing [Gray93], active replication [Birman96], and checkpointing/message logging [Huang93]. Transaction processing is popular in the financial industry. In a transactional system, applications usually have a well-defined transaction boundary, such as updating a record. When a fault occurs, both client and server abort the on-going transaction and rollback to a clean state. Active replication usually involves several identical servers running synchronously. It often assumes a deterministic behavior on these servers and requires an atomic broadcast mechanism to synchronize messages. When a failure occurs in one server, the failure is masked and the computation continues as long as there is one server running. No rollbacks are necessary on either client or server.

Checkpointing and message logging is another way to provide fault tolerant services. The state of a server is checkpointed onto backup servers or on stable storage from time to time. The received messages may also be logged for recreating state change. When a failure occurs, the failed server process is stopped. Then, either a backup server is promoted to the primary, or a new process is created and its state is recovered by loading its last checkpoint and replaying its logged messages. Client may notice some delay during a recovery, but no rollback is involved. Many telecommunication applications constantly manage or monitor some physical devices. Our experience shows that checkpointing and message logging is most suitable for this type of applications [Huang95]. To implement checkpoint and messages logging, we need a number of facilities not provided by Windows NT 4.0. They are application monitoring and failure recovery, application checkpoint and message logging, file replication, Windows events log-

[1] This work is sponsored by Lucent Technologies, Inc.

ging and replay, IP packets dispatching, and IP packets re-routing in case of a machine failure. As a result, each application has to implement its own recovery mechanisms. These recovery mechanisms are usually very complex and hence may not be easy to design and implement by application developers. Therefore, it is desirable to provide them as reusable software components.

In Bell laboratories, we have been working on a set of reusable modules for building reliable and fault tolerant applications for the last 6 years. The set of modules is called SwiFT (Software Implemented Fault Tolerance) [Huang93]. SwiFT has been embedded into tens of telecommunication systems to improve system availability and has been licensed to companies such as Tandem Co., etc. It contains a collection of daemon processes and libraries. SwiFT can be used to handle both client-side and server-side error recoveries. The design philosophy of SwiFT is to make the client error recovery as transparent as possible but provide a set of fault tolerance APIs to be embedded into server programs. This philosophy has proven to be a key to the success of the SwiFT since developers in Bell Labs often have access to the source code of server programs but have no control of client programs developed by other companies.

SwiFT was first implemented and applied on UNIX systems (UNIX-SwiFT). More than two years ago, we started porting SwiFT fault tolerance mechanisms to Windows NT (NT-SwiFT). At the beginning, we were not sure if NT provides enough mechanisms and utilities for us to implement all fault tolerance utilities we need. However, after more than two years of NT-SwiFT effort, we concluded that Windows NT does have all the facilities that are needed to implement SwiFT on Windows NT although some of the NT-SwiFT implementations are quite complex. In this paper, we describe the NT-SwiFT components, their implementation issues and some examples of using NT-SwiFT to enhance applications' reliability and availability. The paper is organized as follows. Section 2 describes NT-SwiFT components. Section 3 discusses some implementation details and issues. Section 4 shows some examples of using NT-SwiFT and performance measurements. Section 5 compares NT-SwiFT with related work. Section 6 concludes the paper.

2. NT-SwiFT Components

As described earlier, NT-SwiFT components can be used in both client and server error recovery. Therefore, we describe NT-SwiFT components in two categories - client components and server components. Please note that since a program could be both a client and a server, all these components can be applied in a program.

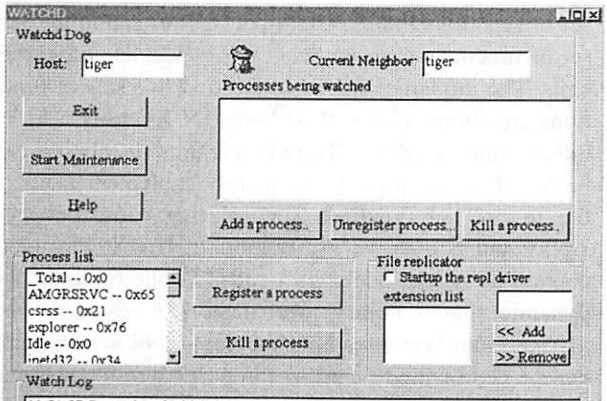
2.1 Components for client error recovery

The design philosophy of client-side recovery components is to make them transparent to client programs. That is, one can embed NT-SwiFT components into client programs without modifying the client source code. A client program may accept a user's keyboard and mouse inputs and, at the same time, talk to one or more server programs running on server machines via communication channels. When a client application fails (either due to a program failure, an OS failure or a machine failure), all input data are lost and all communication channels are broken. Without any fault tolerance facility, the user has to restart the client program, re-establish communication channels and redo all the inputs. This could result in a long recovery time and a frustration of the user. NT-SwiFT provides fault tolerance utilities which (1) detect failure of a client program; (2) automatically restart a client program at failure recovery; (3) re-establish communication channels to server programs; (4) replay all the user inputs and brings the client program back to the state just before the failure occurred.

The first component in NT-SwiFT for the client-side error recovery is *watchd.* Once *watchd* detects a failure, it restarts the application program automatically. If the client application fails too often (more than a threshold given to *watchd*), *watchd* reboots the machine and then restarts the application. The second component is *winckp* which can be used to transparently checkpoint an application program state into a file or another process. In recovery, the checkpointed state is restored back to the client application memory. The third component is *winrecord,* which can be used to log input events from the mouse and keyboard of a client machine. In recovery, the logged input events are replayed to recover the client input data. The last component is *libft* library. *Libft* is used to intercept *winsock* function calls in client applications for checkpointing communication endpoints and logging outgoing messages. In recovery, *libft* re-establishes communication endpoints using the checkpointed information and, if necessary, replays the logged messages.

2.2 Components for server error recovery

On the server side, *watchd* can also be used to detect and recover a server program from a failure. A fault tolerant application process can register its replication strategy to *watchd.* There are two replication strategies that *watchd* supports: hot, and cold. In the hot replication case, *watchd* monitors all replicas of a fault tolerant process; if any replica failure is detected, *watchd* recovers the failed replica on another machine so that the number of replicas (degree of fault tolerance) remains constant. In the cold replication scheme, *watchd* assumes that there is only one active copy of a fault tolerant process; if the active copy fails, *watchd* will first try to recover the failed process on its local machine; if the local recovery fails, *watchd* then migrates the process onto another machine (a fail-over). *Watchd* also provides a few distributed system services such as remote execution, remote file copy, remote status query, etc. Many of these services can be invoked by an application using *libft* APIs. *Watchd* detects two kinds of server failures - hang or crash. To detect a server hang, the server process needs to periodically sends its heartbeats to *watchd.* A server process is considered hung if *watchd* does not receive a heartbeat from the server within a given interval. To send heartbeats to *watchd,* an application can call the *hbeat()* function in *libft* which takes a thread *id* and a timeout value as arguments. To detect a server crash, *watchd* pings the server process periodically. For a hang recovery, *watchd* kills and restarts the hung server process. For a crash recovery, *watchd* first determines the cause of the crash. It can be a machine (including OS failure) or an application program failure. To handle a machine, *watchd* does a fail-over for the server application by either bringing up a cold copy of the server application on another machine or making a warm copy of the server application active. To handle an application program crash failure, *watchd* simply restarts the application. *Watchd* contains a GUI for system configuration as shown in figure [watchd].



Host: tiger

Current Neighbor: tiger

Processes being watched

Process list

```
_Total -- 0x0
AMGRSRVC -- 0x65
csrss -- 0x21
explorer -- 0x76
Idle -- 0x0
inetd32 -- 0x34
```

Watch Log

```
11:21:27:Server thread initializing.
11:21:27:Server thread initialization complete
11:21:27:Polling thread ready.
```

Figure [watchd]: *watchd* GUI

Libft has four major sets of functions for servers:

1 Critical data checkpointing: *Libft* allows an application to select critical data from data segment (e.g. global or static variables) and heap (e.g. data allocated via *malloc()*). The critical data can be saved to a file or to another process on a local or a remote machine, or a protected segment of its own virtual address space. In case of a process crash, the data can be restored into the memory of a newly created process.

2 Communication channel recovery: *Libft* can intercept *winsock* system calls, such as *accept(), listen(), send(), recv()* so that *winsock* communication endpoints and messages can be logged. *Libft* recreates communication endpoints and replays logged messages during a recovery.

3 Requesting services from *Watchd*: *Libft* provides functions for system configurations such as registering a host or a process to be monitored by *watchd.* In addition, it allows an application to send heart-beats to *watchd* and to invoke distributed system services such as remote file copy and status query from *watchd.*

4 Intercepting kernel calls for system handles and file updates: *Libft* can be used to intercept calls that create system handles and that change file contents or attributes. This interception is needed in *winckp* for transparent checkpointing and roll-back recovery [Wang95].

A server program may also create and update files during execution. To make a fail-over possible in a share-nothing environment, component *REPL* can be used to do selective and incremental file replication. By using the *watchd* GUI, a user can specify the types of files that he/she wants to be replicated (see figure [watchd]). For example, a user enters *ppt* in the *"File Replicator"* sub-window in the *watchd* GUI and *REPL* replicates all *powerpoint* files of a machine onto one or more backup machines. *REPL* is typically used in a fail-over environment where files could become unavailable when a machine crashes. *REPL* has also been used for disaster recovery for UNIX applications where a backup machine/disk is located in very far away site.

In a cluster environment, NT-SwiFT provides *ONE-IP* driver (*oneip.sys*) to dispatch and fail-over IP packets. The *ONE-IP* driver provides a single IP image for a cluster of machines. This *ONE-IP* mechanism transparently distributes TCP/IP requests to a set of server machines in a cluster for load balancing and failure recovery [Damani97]. The *ONE-IP* driver can be installed on a set of server machines. A distributed election protocol is used to select one machine as the dispatcher. All server machines in the cluster share the same cluster IP address. In a typical configuration, they (including the dispatcher) run the same applications such as a *web* server, database servers and internet service daemons to provide services. Client applications use the cluster IP address to access a server for services. To achieve load balancing and fault tolerance, the dispatcher picks up the client requests and forwards them to one of the server machines for a service. If the dispatcher fails, *watchd* detects the failure and promotes another machine to become a dispatcher.

3. Implementation Issues

The mechanisms of NT-SwiFT derive from those of UNIX-SwiFT. However, due to the differences between UNIX and NT, their implementations are very different. As mentioned in [Korn97], there are many ways to port UNIX applications to Windows NT. In fact, the first porting effort we tried was to use the UWIN developed by D. Korn in AT&T Labs. However, we later decided to re-implement the NT-SwiFT components from scratch due to the following considerations:

1. Some of the UNIX-SwiFT components such as *REPL, ONE-IP, libckp* and *libft* depend on the UNIX internals. They can not be ported directly by using a library mechanism such as UWIN [Korn97] or a subsystem such as OpenNT [Walli97].
2. We did not want to depend on any third-party software.
3. We wanted to enhance *watchd* with a Windows GUI and threads.
4. To understand how NT application fails, we need to have intimate knowledge of the NT architecture. Re-implementing SwiFT on NT using native NT system services help us to understand the NT internals better.

In this section, we describe how NT-SwiFT components are implemented and the differences in implementations between the UNIX-SwiFT and the NT-SwiFT.

3.1 Watchd

Watchd runs on every machine in a network and uses an adaptive diagnosis protocol [HUANG93] to detect machine failures, i.e., each *watchd* pings its neighbor *watchd*; if its neighbor fails, *watchd* pings its next neighbor and so on. The UNIX version uses three processes to implement *watchd.* The three processes communicate using socket messages and UNIX signals (*SIGUSR1* and *SIGUSR2*). In NT, since there are no corresponding *SIGUSR1* and *SIGUSR2* signals, all functions of NT-*watchd* are implemented in one process with four NT threads. The first thread is the polling thread to detect failures; the second one is the GUI thread for system configuration and display; the third one is the service thread that accepts requests from applications and from other *watchds*; the last thread is the heart-beat thread that accepts application heart beats for a hang detection. Threads are synchronized using semaphores and critical sections. The main advantage of using threads is its low performance overhead – most of the interprocess communication overhead in UNIX watchd modules is removed. However, the main disadvantage of using threads is that self-recovery and fault containment are difficult, if not impossible, to achieve. For example, in UNIX-*watchd* a crash of any module (a process) can be recovered automatically and the failure is transparent to *watchd* clients. However, in NT-*watchd*, any crash of a *watchd* module (a thread) causes the entire *watchd* process to crash.

Watchd uses *OpenProcess()* and *WaitForMultipleObjects()* to detect a application crash (*vs. kill(pid, 0)* and SIGCHLD in UNIX). It uses non-blocking socket calls and time-outs to detect machine failures. *Watchd* detects a process hang by listening to its heartbeats. An application can send its heart beats to *watchd* by calling *hbeat()* functions in *libft*.

3.2 Libft

Libft contains three sets of functions – the first set is for dynamic memory allocation and recovery, the second set of functions is for system configuration and the last set of functions is to intercept *winsock* calls and kernel calls. The implementations of the first two sets of functions are almost identical on both UNIX and NT. More information on *libft* APIs and their implementation can be found in [Huang93]. However, implementations of the last set of functions (intercepting calls) between UNIX and NT are very different. In UNIX, the interception of system and socket calls is done by using the dynamic shared library mechanism (i.e. *dlopen()* and *dlsym()*). On Windows NT, interception of system calls is achieved by modification of import address tables and by the library injection mechanisms [Richter97-18]. To checkpoint and recover kernel states, NT-SwiFT has to intercept all NT calls which create file handles, process handles, thread handles, socket handles and windows handles. It also has to intercept socket calls for messages logging and replay and file system calls which change files contents and attributes. A complete list of kernel and *winsock* calls intercepted by *libft* is illustrated in Table 1.

3.3 REPL

REPL implementation includes one module to intercept file system calls and three daemon processes for sending messages and replaying file system calls. The implementations of the daemon processes are very similar to the UNIX ones. However, the facilities for intercepting file system calls are very different. On UNIX, we use the dynamic shared library mechanism (*dlopen()* and *dlsym()*) to intercept and replay file system calls. On NT, we implement a filter driver, named *REPL.sys*, to intercept file system calls. When a specified type of file is changed, REPL driver intercepts the changes and sends messages to remote backup machines. REPL daemons on remote backup machines then replay the changes to update the files. REPL daemons are all user-level processes, which send I/O messages, log I/O messages and replay I/O messages between the primary host and the backup host. These daemon processes handle link failures, machine failures, I/O failures on the backup machine, messages lost, etc. so that the replicated files are consistent as long as they can be accessed. *Libft* also uses REPL modules to make checkpoint files replicated on all backup machines.

3.4 Winckp

Winckp is a utility program that provides snapshot and rollback functions to an application in a transparent way. Winckp deals with executable files and no source code is needed. *Winckp* starts the application by *CreateProcess()* and obtains its process handle and the thread handle of its main thread. The GUI interface of *Winckp* allows a user to take snapshot of an application or roll back the memory of an application. To take a snapshot, *Winckp* suspends the main thread and stores the thread context and memory content into a checkpoint file. The thread context is obtained and restored using *GetThreadContext()* and *SetThreadContext()*. The memory image is obtained and restored using *ReadProcessMemory()* and *WriteProcessMemory()*. *Winckp* determines the address and the amount of memory needed to be saved. An NT process has about 2GB of private address space, ranging from 0x00010000 through 0x7FFEFFFF [Richter97-3]. Note that not every region in this space needs to be saved. The *VitualQueryEx()* system call allow us to examine the space region by region. A memory region is necessary to be saved if its write access is enabled and if its physical storage is committed [Richter97-5]. *Winckp* also stores the MEMORY_BASIC_INFORMATION structure along with each memory region. During a rollback operation, the application main thread is suspended. *Winckp* reads the thread context from the checkpoint file and calls *SetThreadContext()*. It then calls *WriteProcessMemory()* to restore the memory content.

To recover an application process from a failure, *winckp* not only has to restore the process memory content but also has to recreate all the handles that were owned by the process before the recovery. To checkpoint and recover kernel states, *winckp* uses the *libft* interception facilities to intercept all NT calls which create file handles, process handles, thread handles, socket handles, such as *CreateProcess()*, *CreateFile()*, *CreateThread()*, etc. *Winckp* records each handle value and the parameters that are used to create the handle. In recovery, *winckp* recreates all the handles by replaying the calls with the recorded parameters. To make the values of the recovered handles equal to their recorded values, *winckp* uses a different mechanism from the UNIX version (namely *libckp* [Wang95]). In *libckp*, each newly created handle is duplicated to its old value by using the *dup2()* call. In NT, since there is no function that can duplicate a handle to a given handle value, *winckp* uses a loop that keeps duplicating a handle using *DuplicateHandle()* call till the returned handle value is equal to the recorded value. Then, all other handles are closed. This process is repeated till all the handles are created.[2]

Winckp also uses *libft* to intercept file system calls. When an application takes a checkpoint, it has not only to save its memory contents but also its file contents and attributes. When the application rolls back to its previous checkpointed state, it has to undo all file updates after the last checkpoint as well as restore its memory content. The file roll-back mechanism uses the *libft* interception routines as described in [Wang95].

3.5 Winrecord

In *winrecord*, we are primarily interested in system events related to keyboard strokes and mouse inputs. Win32 subsystem provides a hook that allows a user application to monitor system events such as keyboard strokes, window messages, debugging information, etc., and to react to these events through a user-defined callback procedure. User application may specify those system events of interest and install the corresponding callback procedures via the Win32 API. *Winrecord* captures those events by calling *SetWindowsHookEx()* with flag WH_JOURNALRECORD, and all keyboard

[2] This mechanism does not work for Windows handles.

events and mouse events are copied from the Win32 system's message queue to our callback procedure. These events are kept in a temporary file. To replay, we insert these events one after the other in their timestamp order back to Win32 system message queue by installing the WH_JOURNALPLAYBACK callback procedure. The Win32 system temporarily disables the inputs from keyboard and mouse when the WH_JOURNALPLAYBACK callback procedure is installed. It executes only the events fed from the callback procedure until our event log is up and the WH_JOURNALPLAYBACK callback procedure is uninstalled.

3.6 *ONE-IP* driver

The *ONE-IP* driver is an NDIS (Network Driver Interface Specification) intermediate driver. It is sitting between transport drivers and NDIS NIC (Network Interface Card) mini-ports. The driver is installed on every machine in the cluster. Our design works in the following way: all the client request packets are first sent to the dispatcher machine and the dispatcher machine selects a server from the cluster and forwards the packet to that server. A problem is that all machines share the cluster IP address. In order for a packet to reach the dispatcher, only the dispatcher should reply ARP requests for the cluster IP. In our implementations, when the immediate driver on a server machine receives an ARP request packet for the cluster IP address, if it is not the dispatcher, it discards the ARP packet.

On the dispatcher machine, when the NIC driver receives a packet, it calls the *ReceiveHandler()* in the transport interface of the *ONE-IP* intermediate driver. The *ReceiveHandler()* examines the packet. If the packet is from a client request, it contains an Ethernet packet header and an IP packet in the lookahead buffer. If the destination address of the IP packet matches the cluster IP address, a server is selected based on a hash value of the client IP address (source address in the IP packet). The Ethernet packet header is then modified in the following way: the source MAC address is changed to the dispatcher's MAC address; the destination MAC address is changed to the selected server's MAC address. The packet is then sent to the NIC driver by *NdisSend()* call and reaches the selected server. Since a dispatcher can also be servicing requests, if the dispatcher itself is selected, then the packet is passed up to the protocol driver without modifications.

One desired feature for the *ONE-IP* driver is the capability to dynamically reconfigure the dispatching hash function, the cluster IP address or the cluster size. To achieve this, we create a logical device in the *ONE-IP* driver by *IoCreateDevice()* and expose a device name in the NT object namespace, *\\device\oneip*. A user-level program can change parameters in the *ONE-IP* driver by first obtaining a handle to the logical device by *CreateFile()* with the device name and then issuing a *DeviceIoControl()* via the handle.

To make the *ONE-IP* dispatching mechanism fault tolerant, we integrate *ONE-IP* driver with the *watchd* daemon. As mentioned earlier, *watchd* runs on every machine in a SwiFT domain. When the first *watchd* comes up in the domain, it makes its own *ONE-IP* driver the primary dispatcher by calling the *DeviceIoControl (sys_handle, SET_PRIMARY,...)*. When the dispatcher machine fails, the neighboring *watchd* detects the failure and set its *ONE-IP* driver the new primary dispatcher.

The UNIX implementation of *ONE-IP* is done in the NetBSD kernel [Damani97] [Wang 97]. The dispatcher runs our modified kernel and is configured to run in the routing mode. The main kernel modifications are in the IP forwarding layer ([Wright 95] p.222). We modified the `ip_forward()` routine so that the selected server's IP address is used as the next hop for the packet.

Since the UNIX implementation involves changing kernel code, it is difficult to port to a system where the kernel source code is unavailable. On the other hand, the NDIS driver approach on NT is much easier to be adopted into a product.

4. Applications and Overhead

We are currently working with a few projects in Lucent Technologies to embed NT-SwiFT in their systems to improve their fault tolerance and availability. In one project, the system uses NT-SwiFT to detect application failures such as process crashes and hangs. Once a failure is detected, *watchd* stops the process and restart the process. If a process fails too many times in a given interval, *watchd* then automatically reboots the NT machine and restarts the application. In another project, we are using *watchd* and *libft* to provide a warm backup scheme for a switch prototype implemented on Windows NT where processes on the primary board checkpoint their critical states to the backup processes on a backup board whenever necessary. When a failure is detected, *watchd* makes the backup board the primary by changing a flag in the shared memory of the board.

In a normal situation, *watchd* polls applications and machines every 10 seconds and one polling takes about 10 milliseconds on a Pentium 180MHz machine. By polling, *watchd* increases the CPU utilization by about 4%. *Libft* overhead depends on the frequency of checkpointing and message logging. In one study, it showed 5 to 10% increases for the service time of a server program when checkpoint and message logging were used. REPL overhead also depends on the intensity of I/O write operations. One study showed 14% decrease of I/O throughputs when using REPL in replicating files for a disaster recovery.

5. Comparison with Related Work

Some of the NT-SwiFT functions are also provided by a few commercial NT cluster mechanisms. A survey on NT clustering solutions can be found in [NT-CLUSTER]. Examples of NT clustering solutions are Microsoft MSCS, Tandem CAS, Marathon Endurance, Apcon PowerSwitch, NCR LifeKeeper, Veritas FirstWatch, Octopus HA+, etc. These commercial NT cluster products provide basic fail-over and detection capabilities. Some of them also provide file replication or disk-mirroring facilities for persistent data recovery. However, there are at least three major differences between NT-SwiFT and these clustering tools:

1. The fundamental design philosophy is different between NT-SwiFT and these commercial tools. Most of these tools assume application programs can not be changed and therefore all the recovery mechanisms have to be completely transparent to application programs. Consequently, these clustering tools provide either no application APIs or a very small set of APIs to be embedded into application programs. Our design philosophy considers the recovery mechanisms into two categories: client recovery and server recovery. We also think that the client error recovery mechanisms have to be transparent to the client application programs. However, we believe that a truly fault tolerant server application has to be enhanced with fault tolerance APIs. Therefore, a large part of our effort is to design and implement a set of fault tolerance APIs for the server application developers[3]. As a result, the APIs provided by *libft* are more powerful and complete than those provided by these commercial clustering tools. These fault tolerance APIs also have to interact with other components in SwiFT such as *watchd, REPL, winrecord, winckp* and *ONE-IP*. Therefore, an integration of all fault tolerance components is a must but none of the commercial clustering tools provides such integration.
2. Most of these clustering tools assume transaction model for error recovery while our focus is on the checkpoint and roll-back recovery. As a result, none of these tools integrate roll-back recovery mechanisms such as process checkpoint, events/messages logging and replay, etc. into their recovery mechanisms. Without an integrated solution, application developers may have to design and implement a lot of recovery routines into their programs.
3. NT-SwiFT provides facilities to do application rejuvenation [Garg96][4], IP requests dispatching, process migration and load balancing. As a result, NT-SwiFT can not only increase application availability but also improve application robustness, performance and scalability.

6. Concluding Remarks and Future work

The goal of NT-SwiFT research is to understand the fault-tolerance and high availability requirements of applications running on NT and to create generic and reusable components that can facilitate the development of these applications. We have described components including *watchd* for process failure detection and recovery, *libft* for critical data checkpointing, communication messages logging and recovery, *REPL* for on-line incremental file replication and disaster recovery, *winckp* for transparent process checkpointing, *winrecord* for mouse and keyboard events logging and replaying, and *ONE-IP* for IP packets dispatching, fail-over and re-routing. We have demonstrated that leveraging specific facilities on Windows NT such as filter drivers, intermediate drivers, library injection and memory management routines makes the implementation of some fault tolerance mechanisms easier on Windows NT than on UNIX.

Currently, we are working on enhancing the NT-SwiFT to deal with process thread failure detection and recovery, incremental state checkpoint to remote processes, integration of the NT-SwiFT with some middle-ware

[3] Note that except some functions in *libft*, all other components in NT-SwiFT can be used transparently with application programs.

[4] Application rejuvenation is a mechanism which monitors applications behaviors, predicts applications failures and rejuvenates unhealthy applications even before they actually fail.

tools such as CORBA and DCOM, dynamic process migration for load balancing, intercepting calls in other DLLs such as *advapi32.dll, user32.dll, GDI32.dll*, etc., and the compatibility of NT-SwiFT with other popular commercial clustering tools such as MSCS.

Acknowledgements: Gaurav Suri and Yi-Min Wang implemented the first prototype of *watchd* and *libft* in NT-SwiFT. Recently, Woei-Jyh Lee joined our NT-SwiFT team and contributed in the porting of *ONE-IP* driver and *watchd*. The authors would also like to thank the users of the NT-SwiFT who constantly provide ideas for improvements and Dave Korn for his help in using UWIN and comments on this paper.

References:

[Birman96] Kenneth P. Birman, "Building Secure and Reliable Network Applications", Manning Publication Co. 1996.

[Damani97] Damani, O. P., Chung, P.-Y., Huang, Y., Kintala, C. M., and Wang, Y-M., "*ONE-IP*: Techniques for Hosting a Service on a Cluster of Machines", *Sixth International World Wide Web Conference (WWW6)*, Santa Clara, pp. 735-743, Apr. 1997.

[DDK-NDIS] "Network Drivers", Windows NT 4.0 DDK, Microsoft MSDN Library.

[Huang93] Huang, Y. and Kintala, C. "Software Implemented Fault Tolerance", *Proceedings of the 23rd IEEE Fault Tolerant Computing Symposium (FTCS23)*, Toulous, France, June 1993, Pages 2-10.

[Huang95] Y. Huang and Y. Wang, ''Why optimistic message logging has not been used in telecommunication", *Proceedings of the 25th IEEE Fault Tolerant Computing Symposium (FTCS25), Pasadena*, California, page 459-463, 1995.

[Garg96] S. Garg and Y. Huang and K. Trivedi and C. Kintala, ''Minimizing Completion Time of a Program by Checkpointing and Rejuvenation",ACM SIGMETRICS 96, Philadelphia, PA, pages 252-261, May, 1996.

[Gray93] J. Gray and A. Reuter, "Transaction Processing: Concepts and Techniques", Morgan Kaufmann Publishers, 1993.

[Richter96-3] J. Richter, "Processes", Chapter 3, in *Advanced Windows,* Ed. 3, pp.33-72, Microsoft Press, 1996.

[Korn97] D. Korn, “UWIN – UNIX for Windows”, *Proceedings of Usenix Windows NT Workshop*, Seattle, Washington, pp. 133-145, 1997.

[NTCLUSTER] "Lab Reports: Clustering Solutions for Windows NT", Windows NT Magazine, pp.54-95, June 1997.

[Richter97-5] J. Richter, "Win32 Memory Architecture", Chapter 5, in *Advanced Windows*, Ed. 3, pp.115 - 144, Microsoft Press, 1997.

[Richter97-18] J. Richter, "Breaking Through Process Boundary Wall", Chapter 18, in *Advanced Windows*, Ed. 3, pp.899-970, Microsoft Press, 1997.

[Walli97] S. R. Walli, “OpenNT: UNIX Application Portability to Windows NT via an Alternative Environment Subsystem”, *Proceedings of Usenix Windows NT Workshop*, Seattle, Washington, pp. 123-132, 1997.

[Wang95] Y. Wang and Y. Huang and K. Vo and E. Chung and C. Kintala, “Checkpoint and its applications”, *Proceedings of the 25th IEEE Fault Tolerant Computing Symposium,* Pasadena, California, pp. 22-31, 1995.

[Wang97] Y.-M. Wang, O. P. Damani, P. E. Chung, Y. Huang and C. M. Kintala, Web Server Clustering with Single-IP Image: Design and Implementation", in *Proc. Int. Symp. on Multimedia Information Processing*, Dec. 1997, also in http://www.research.att.com/~ymwang/papers/newONE-IP.htm

WINDOWS SOCKET (Wsock32.dll)	accept, bind, closesocket, connect, ioctlsocket, listen, setsockopt, shutdown, socket, send, recv, recvfrom, sendto
File Operations (Kernel32.dll)	CopyFileA, CopyFileExA, CopyFileExW, CopyFileW, CreateFileA, CreateFileW, DeleteFileA, DeleteFileW, MoveFileA, MoveFileW, MoveFileExA, MoveFileExW, OpenFile, ReadFile, ReadFileEx, ReadFileScatter, SetFilePointer, UnlockFile, UnlockFileEx, WriteFile, WriteFileEx, WriteFileGather.
Directory (Kernel32.dll)	CreateDirectoryA, CreateDirectoryExA, CreateDirectoryExW, CreateDirectoryW, RemoveDirectoryA, RemoveDirectoryW, SetCurrentDirectoryA, SetCurrentDirectoryW.
Process and Thread (Kernel32.dll)	CreateRemoteThread, CreateThread, CreateProcessA, CreateProcessW, ExitProcess, ExitThread, OpenProcess, TerminateProcess, TerminateThread.
Event (Kernel32.dll)	CreateEventA, CreateEventW, OpenEventA, OpenEventW, ResetEvent, SetEvent.
NamedPipe (Kernel32.dll)	ConnectNamedPipe, CreateNamedPipeA, CreateNamedPipeW, DisconnectNamedPipe, SetNamedPipeHandleState, WaitNamedPipeA, WaitNamedPipeW.
MailSlot (Kernel32.dll)	CreateMailslotA, CreateMailslotW, SetMailslotInfo.
Mutex (Kernel32.dll)	CreateMutexA, CreateMutexW, OpenMutexA, OpenMutexW, ReleaseMutex
Semaphore (Kernel32.dll)	CreateSemaphoreA, CreateSemaphoreW, OpenSemaphoreA, OpenSemaphoreW, ReleaseSemaphore
CriticalSection (Kernel32.dll)	EnterCriticalSection, InitializeCriticalSection, InitializeCriticalSectionAndSpinCount, LeaveCriticalSection
DLL Library (Kernel32.dll)	FreeLibrary, FreeLibraryAndExitThread, LoadLibraryA, LoadLibraryExA, LoadLibraryExW, LoadLibraryW
Other handles (Kernel32.dll)	CloseHandle, DuplicateHandle, SetHandleCount, SetHandleInformation

Table 1. A summary of NT system calls that are intercepted in NT-SwiFT.

A Thread Performance Comparison:
Windows NT and Solaris on A Symmetric Multiprocessor

Fabian Zabatta and Kevin Ying
Logic Based Systems Lab
Brooklyn College and CUNY Graduate School
Computer Science Department
2900 Bedford Avenue
Brooklyn, New York 11210
{fabian, kevin}@sci.brooklyn.cuny.edu

Abstract

Manufacturers now have the capability to build high performance multiprocessor machines with common PC components. This has created a new market of low cost multiprocessor machines. However, these machines are handicapped unless they have an operating system that can take advantage of their underlying architectures. Presented is a comparison of two such operating systems, Windows NT and Solaris. By focusing on their implementation of threads, we show each system's ability to exploit multiprocessors. We report our results and interpretations of several experiments that were used to compare the performance of each system. What emerges is a discussion on the performance impact of each implementation and its significance on various types of applications.

1. Introduction

A few years ago, high performance multiprocessor machines had a price tag of $100,000 and up, see [16]. The multiprocessor market consisted of proprietary architectures that demanded a higher cost due to the scale of economics. Intel has helped to change that by bringing high performance computing to the mainstream with its Pentium Pro (PPro) processor. The PPro is a high performance processor with built in support for multiprocessing, see [4]. This coupled with the low cost of components has enabled computer manufacturers to build high performance multiprocessor machines at a relatively low cost. Today a four processor machine costs under $12,000.

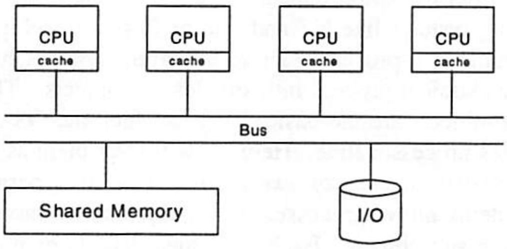


Figure 1: A basic symmetric multiprocessor architecture.

1.1 Symmetric Multiprocessing

Symmetric multiprocessing (SMP) is the primary parallel architecture employed in these low cost machines. An SMP architecture is a tightly coupled multiprocessor system, where processors share a single copy of the operating system (OS) and resources that often include a common bus, memory and an I/O system, see Figure 1. However, the machine is handicapped unless it has an OS that is able to take advantage of its multiprocessors. In the past, the manufacturer of a multiprocessor machine would be responsible for the design and implementation of the machine's OS. It was often the case, the machines could only operate under the OS provided by the manufacturer. The machines described here are built from common PC components and have open architectures. This facilitates an environment where any software developer can design and implement an OS for these machines. Two mainstream examples of such operating systems are Sun's Solaris and Microsoft's Windows NT. They exploit the power of multiprocessors by incorporating *multitasking* and *mul-*

tithreading architectures. Their implementations are nevertheless very different.

1.2 Objects Of Execution

Classic operating systems, such as UNIX, define a *process* as an object of execution that consists of an address space, program counter, stack, data, files, and other system resources. Processes are individually dispatched and scheduled to execute on a processor by the operating system *kernel*, the essential part of the operating system responsible for managing hardware and basic services. In the classic case, a multitasking operating system divides the available CPU time among the processes of the system. While executing, a process has only one unit of control. Thus, the process can only perform one task at a time.

In order to exploit concurrency and parallelism, operating systems like NT and Solaris further develop the notion of a process. These operating systems break the classical process into smaller sub-objects. These sub-objects are the basic entity to which the OS allocates processor time. Here we will refer them as *kernel-level objects of execution.* Current operating systems allow processes to contain one or more of these sub-objects. Each sub-object has its own *context*[1] yet it shares the same address space and resources, such as open files, timers and signals, with other sub-objects of the same process. The design lets the sub-objects function independently while keeping cohesion among the sub-objects of the same process. This creates the following benefits.

Since each sub-object has its own context each can be separately dispatched and scheduled by the operating system kernel to execute on a processor. Therefore, a process in one of these operating systems can have one or more units of control. This enables a process with multiple sub-objects to overlap processing. For example, one sub-object could continue execution while another is blocked by an I/O request or synchronization lock. This will improve throughput. Furthermore with a multiprocessor machine, a process can have sub-objects execute concurrently on different processors. Thus, a computation can be made parallel to achieve speed-up over its serial counterpart. Another benefit of the design arises from sharing the same address space. This allows sub-objects of the same process to easily communicate by using shared global variables. However, the sharing of data requires synchronization to prevent simultaneous access. This is usually accomplished by using one of the synchronization variables provided by the OS, such as a *mutex.* For general background information on synchronization variables see [14], for information on Solaris's synchronization variables see [1, 5, 12, 13], and [7, 10, 15] for Windows NT's synchronization variables.

2. Solaris's and Windows NT's Design

Windows NT and Solaris further develop the basic design by sub-dividing the kernel-level objects of execution into smaller *user-level objects of execution.* These user-level objects are unknown to the operating system kernel and thus are not executable on their own. They are usually scheduled by the application programmer or a system library to execute in the context of a kernel-level object of execution.

Windows NT and Solaris kernel-level objects of execution are similar in several ways. Both operating systems use a priority-based, time-sliced, preemptive multitasking algorithm to schedule their kernel-level objects. Each kernel-level object may be either interleaved on a single processor or execute in parallel on multiprocessors. However, the two operating systems differ on whether user-level or kernel-level objects should be used for parallel and concurrent programming. The differences have implications on the overall systems' performances, as we will see in later sections.

2.1 NT's Threads and Fibers

A *thread* is Windows NT's smallest kernel-level object of execution. Processes in NT can consist of one or more threads. When a process is created, one thread is generated along with it, called the *primary thread.* This object is then scheduled on a system wide basis by the kernel to execute on a processor. After the primary thread has started, it can create other threads that share its address space and system resources but have independent contexts, which include execution stacks and thread specific data. A thread can execute any part of a process' code, including a part currently being executed by another thread. It is through threads, provided in the Win32 application programmer interface (API), that Windows NT allows programmers to exploit the benefits of concurrency and parallelism.

[1] This refers to its state, defined by the values of the program counter, machine registers, stacks, and other data.

A *fiber* is NT's smallest user-level object of execution. It executes in the context of a thread and is unknown to the operating system kernel. A thread can consist of one or more fibers as determined by the application programmer. Some literature [1,11] assume that there is a one-to-one mapping of user-level objects to kernel-level objects, this is inaccurate. Windows NT does provide the means for many-to-many scheduling. However, NT's design is poorly documented and the application programmer is responsible for the control of fibers such as allocating memory, scheduling them on threads and preemption. This is different from Solaris's implementation, as we shall see in the next section. See [7, 10] for more details on fibers. An illustrative example of NT's design is shown in Figure 2.

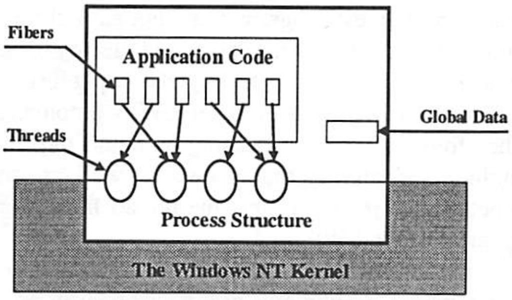


Figure 2: The relationships of a process and its threads and fibers in Windows NT.

2.2 Solaris's LWPs and Threads

A *light weight process* (LWP) is Solaris's smallest kernel-level object of execution. A Solaris process consists of one or more light weight processes. Like NT's thread, each LWP shares its address space and system resources with LWPs of the same process and has its own context. However, unlike NT, Solaris allows programmers to exploit parallelism through a user-level object that is built on light weight processes. In Solaris, a *thread* is the smallest user-level object of execution. Like Windows NT's fiber, they are not executable alone. A Solaris thread must execute in the context of a light weight process. Unlike NT's fibers, which are controlled by the application programmer, Solaris's threads are implemented and controlled by a system library. The library controls the mapping and scheduling of threads onto LWPs automatically. One or more threads can be mapped to a light weight process. The mapping is determined by the library or the application programmer. Since the threads execute in the context of a light weight process, the operating system kernel is unaware of their existence. The kernel is only aware of the LWPs that threads execute on. An illustrative example of this design is shown in Figure 3.

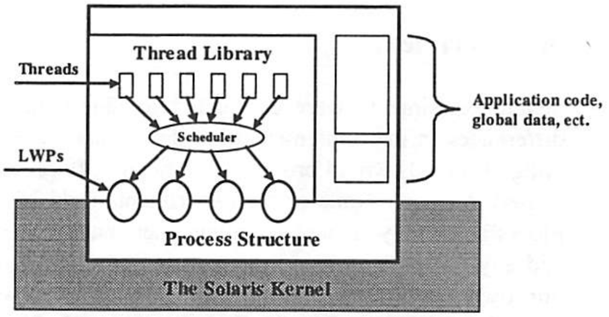


Figure 3: The relationships of a process and its LWPs and threads in Solaris.

Solaris's thread library defines two types of threads according to scheduling. A *bound* thread is one that permanently executes in the context of a light weight process in which no other threads can execute. Consequently, the bound thread is scheduled by the operating system kernel on a system wide basis. This is comparable to an NT thread.

An *unbound* thread is one that can execute in the context of any LWP of the same process. Solaris uses the thread library for the scheduling of these unbound threads. The library works by creating a pool of light weight processes for any requesting process. The initial size of the pool is one. The size can be automatically adjusted by the library or can be defined by the application programmer through a programmatic interface. It is the library's task to increase or decrease the pool size to meet the requirements of an application. Consequently, the pool size determines the concurrency level (CL) of the process. The threads of a process are scheduled on a LWP in the pool, by using a priority based, first-in first-out (FIFO) algorithm. The priority is the primary algorithm and FIFO is the secondary algorithm (within the same priority). In addition, a thread with a lower priority may be preempted from a LWP by higher priority thread or by a library call. Here we will only consider threads of the same priority without preemption. Thus, the scheduling algorithm is solely FIFO. In this case, once a thread is executing it will execute to completion on a light weight process unless it is blocked or preempted by the user. If blocked, the library will remove the blocked thread from the LWP and schedule the next thread from the queue on that LWP. The new thread will execute on the LWP until it has completed or been blocked. The scheme will then continue in the same manner. For more information on Solaris's design and implementation, see [1, 5, 12, 13].

3. Experiments

Seven experiments were conducted to determine if differences in the implementation, design and scheduling of threads would produce significant differences in performance. None of the experiments used NT's Fibers since they require complete user management and any comparison using them would be subject to our own scheduling algorithms. Furthermore, we wanted to test each system's chosen thread API. Thus we chose to compare the performance of NT's threads to three variations of Solaris's threads (bound, unbound, and restricted concurrency). Although it may seem unfair to compare NT's kernel implementation to Solaris's user implementation, it is fair because Solaris's implementation is not purely user based. Embedded in its design are kernel objects, LWPs. Therefore, like the NT case, the OS kernel is involved in scheduling. Furthermore, the comparison is between each operating system's chosen thread model and thus we are comparing models that each system has specifically documented for multithreading. The models do have fundamental differences, yet they still warrant a comparison to determine how each could effect different aspects of performance. The conducted experiments tried to achieve this by measuring the performance of each system under different conditions. In some cases, the experiments tried to simulate the conditions of real applications such the ones found in client/server computing and parallel processing. The seven experiments were:

1. Measure the maximum number of kernel threads that could be created by each system (Section 3.2).
2. Measure the execution time of thread creation (Section 3.2).
3. Measure the speed of thread creation on a heavily loaded system (Section 3.2).
4. Determine how each operating system's thread implementation would perform on processes with CPU intensive threads that did not require any synchronization (Section 3.3).
5. Determine how each operating system's thread implementation would perform on processes with CPU intensive threads that required extensive synchronization (Section 3.4).
6. Determine the performance of each operating system's implementation on parallel searches implemented with threads (Section 3.5).
7. Determine the performance of each operating system's implementation on processes with threads that had bursty processor requirements (Section 3.6).

To restrict the concurrency of Solaris's threads in the experiments, unbound threads were created and their concurrency was set to the number of processors, noted by **(CL = 4)** in the tables. In theory, this created a LWP for each processor and imposed on the thread library to schedule the unbound threads on the LWPs. To use Solaris unbound threads, threads were created without setting the concurrency level. Thus, only one LWP was made available to the thread library, and the test program could not exploit the multiprocessors. However, the thread library is documented [11, 12, 13] as having the capability to increase or decrease the pool of LWPs automatically. Therefore, any processes using unbound threads, including processes that contain with restricted concurrency, indirectly test the thread library's management of the LWP pool.

Our reported benchmarks are in seconds for an average of 10 trials. In most cases, the standard deviations (σ) for trails were less than 0.15 seconds. We only report σ, when a trial's $\sigma \geq 0.2$ seconds.

3.1 Parameters

We acknowledge the arguments on the stability of benchmarks presented in [2]. Thus, we take every precaution to create a uniform environment. For all experiments, the default priority was used for each system's kernel-level objects of execution. Experiments were performed on the same hardware, a four PPro SMP machine with 512 megabytes of RAM and a four-gigabyte fast and wide SCSI hard drive. At any one time, the machine solely operated under Solaris version 2.6 or Windows NT Server version 4.0. This was implement by a dual boot to facilitate easy switching between the OSs. Each operating system used its native file system. There were no other users on the machine during experiments. The "same compiler", GNU gcc version 2.8.1, was used to compile the test programs for both operating systems. Originally, this was done to reduce any variances in execution time that could be attributed to different compilers. However, later we compiled the test programs with each system's native compiler (Visual C++ 5.0

and SUN C Compiler 4.2) and found no significant differences in performance. In order to maintain a standard format, we chose to only report the results from the gcc compilations. Note, all test programs were compiled with the options: *-O3 -mpentiumpro -march=pentiumpro*. These options generate the highest level of performance optimizations for a Pentium Pro.

3.2 Thread Creation

The first experiment measured the maximum number of kernel-level objects of execution each operating system could create, since neither system clearly documents the limit. The experiment was performed by repeatedly creating threads (bound in the Solaris case) that suspended after creation. At some given point, the tested operating system would fail trying to create a thread because it had exhausted resources or had reached the preset limit.

Description	NT	Solaris
# of Threads Created	9817	2294
Memory Usage	68MB	19MB
Execution Time (sec.)	24.12	2.68

Table 1: Comparison of the maximum number of threads allowable.

Table 1 shows the test program executing on Solaris failed after 2294 requests to create bound threads. At the time of termination, the process had only used 19 megabytes of memory. The Windows NT test program created 9817 threads before failing. At that time, it had used approximately 68 megabytes of memory. In addition, the table shows the amount of time required to create the threads.

The second experiment, shown in Table 2, measured the speed of thread creation on both systems. The table shows Solaris bound threads and NT threads had similar performances. The similar performance can be attributed to the fact that each OS required system calls for thread creation. In the case of Solaris, this was done indirectly through the thread library. The library was required to make a system call to create an LWP for each bound thread. In addition as expected, Solaris's unbound thread creation outperformed NT's. In this case, Solaris's thread creation required a simple library call, while NT's required a system call. It is also worth noting that the Solaris restricted concurrency case (CL=4) was only marginally slower than the Solaris unbound case. This was because only four LWPs were needed. Thus, only four system calls were required to create the threads.

Threads	NT Time	Solaris Time		
		Bound	CL=4	Unbound
100	0.11	0.08	0.07	0.06
200	0.17	0.15	0.11	0.09
500	0.37	0.37	0.24	0.22
1000	0.74	0.81	0.49	0.45
2000	1.90	2.07	1.12	0.98

Table 2: Comparison of thread creation speed.

The third experiment also involved the creation of threads. The experiment measured the speed of thread creation while other threads were executing CPU intensive tasks. The tasks included several integer operations such as addition, subtraction, and multiplication. This imposed on each OS to create threads on a heavily loaded system. The number of threads created was varied. Table 3 shows how long it took to create a collection of threads in each system.

Threads	NT		Solaris	
	Time	σ	Time	σ
16	3.65	0.29	0.33	0.04
32	5.72	0.34	0.52	0.14
64	12.56	0.43	0.91	0.22
128	146.74	18.77	1.98	0.39

Table 3: Comparison of the performance of processes that create CPU intensive threads.

The experiment showed that the Solaris version of the test program created threads much faster than the NT version. This can be attributed to each systems multitasking scheduling algorithm. Although, the algorithms are similar in design, priority differences exist. Solaris's algorithm was fair with respect to newly created LWPs, while NT scheduling algorithm gave priority to currently executing threads. Thus in the case of NT, requests for thread creation took longer because of the heavily loaded system. We found this to be characteristic of NT's scheduling algorithm. In various cases, executing several CPU-bound threads severely reduced the responsiveness of the system. Microsoft documents this behavior in [7]. Also, in both the Solaris and the NT cases, as the number of threads increased, the thread creation time became less predictable. This was especially true in the NT case, $\sigma = 18.77$ seconds when 128 threads were used.

3.3 No Synchronization

The fourth experiment determined how each operating system's thread implementation would perform on processes that created CPU intensive threads (with identical workloads) that did not require any synchronization. The experiment was performed by executing a process that created threads, where each thread had a task to perform that required a processor for approximately 10 consecutive seconds. A thread would perform its task and then terminate. After all the threads terminated, the creating process would terminate. Table 4 shows how long it took processes to complete in each system.

Threads	NT Time	Solaris Time		
		Bound	CL=4	Unbound
1	10.11	10.06	10.06	10.06
4	10.13	10.13	10.12	40.18
8	20.32	20.53	20.26	80.35
16	40.37	40.35	40.52	160.67
32	80.49	80.80	80.73	321.27
64	160.78	161.34	161.49	642.54

Table 4: Comparison of the performance of processes with CPU intensive threads that do not require synchronization.

The experiment showed few differences in performance between NT threads and Solaris bound threads. This suggests that Solaris bound threads are similar to NT threads while performing CPU intensive tasks that did not require synchronization. However, it is worth noting that as the number of CPU intensive threads increased, Windows NT's performance was slightly better.

In Solaris's unbounded and CL=4 cases, the thread library did not increase nor decrease the size of the LWP pool. Therefore, only one LWP was used by the library for the unbounded case. Consequently, the unbound threads took approximately 10*N* time, where *N* was the number of threads used. (Recall each thread performed a 10 second task.) The performance was also reflective of the FIFO algorithm used by library. Another point worth noting is that in Solaris CL=4 case, the performance was equivalent to that of the bound threads, which were optimal. Thus, additional LWPs did not increase the performance. This leads to two observations. First, in the case of equal workloads with no synchronization, peek performance is reached when the amount of LWPs is equal to the number of processors. Second, the time it takes Solaris's thread library to schedule threads on LWP is not a factor in performance.

3.4 Extensive Synchronization

The fifth experiment determined how each operating system's thread implementation would perform on processes that use threads (with identical workloads), which require extensive synchronization. The test program was a slightly altered version of an example from [1] called "array.c". The test program created a variable number of threads that modified a shared data structure for 10000 iterations. Mutual exclusion was required each time a thread needed to modify the shared data. In the general case, this can be implemented with a mutual exclusion object, like a mutex. Both operating systems offer local and global mutual exclusion objects[2]. Windows NT provides two mutual exclusion objects, a *mutex*, which is global, and a *critical section*, which is local. Solaris only provides a *mutex*. However, an argument can be passed to its initialization function, to specify its scope. We thus chose to compare each system's local and global mutual exclusion objects. Tables 5 and 6 shows the execution times for processes to complete in each system.

The results show NT out performs Solaris when using local synchronization objects, while Solaris out performs NT when using global synchronization objects. In addition, the experiment showed large differences in the performance of NT's mutex in comparison to its critical section, and few differences in performance of Solaris local mutex in comparison to its global mutex. The poor performance of NT's mutex was directly attributed to its implementation. NT's mutex is a kernel object that has many security attributes that are used to secure its global status. NT's critical section is a simple user object that only calls the kernel when there is contention and a thread must either wait or awaken. Thus, its stronger performance was due to the elimination of the overhead associated with the global mutex.

The Solaris case CL = 4 outperformed both bound and unbound Solaris cases. This was due to a reduction in contention for a mutex. This reduction was caused by the LWP pool size. Since the pool size was four, only four threads could execute concurrently. Consequently, only four threads could con-

[2] This refers to the scope of the synchronization object, where local refers to a process scope and global refers to a system scope.

tend for the same mutex. This reduced the time threads spent blocked, waiting for a mutex. Furthermore, when a thread was blocked, the thread library scheduled another thread on the LWP of the blocked thread. This increased the throughput of the process.

Threads	NT Time	Solaris Time		
	Critical	Local Mutex		
	Section	Bound	CL=4	Unbound
250	1.04	1.20	1.13	2.69
500	2.49	2.93	2.56	5.98
750	3.76	5.16	4.37	9.63
1000	4.93	8.18	6.43	13.89
2000	9.89	24.85	17.84	35.38

Table 5: Comparison of the performance of processes with threads that required extensive synchronization using local/intra-process syncronization objects.

Threads	NT Time	Solaris Time		
	Mutex	Global Mutex		
		Bound	CL=4	Unbound
250	10.84	1.18	1.20	2.68
500	25.58	2.69	2.74	5.94
750	37.78	4.80	4.60	9.59
1000	49.73	7.79	6.95	13.73
2000	99.15	24.98	19.75	34.89

Table 6: Comparison of the performance of processes with threads that require extensive synchronization using global/inter-process synchronization objects.

3.5 Parallel Search

The sixth experiment determined how each operating system's thread implementation would perform on the execution of a parallel search implemented with threads that required limited synchronization. Here we explored the classic symmetric traveling salesman problem (TSP). The problem is defined as follows:

> Given a set of n nodes and distances for each pair of nodes, find a roundtrip of minimal total length visiting each node exactly once. The distance from node i to node j is the same as from node j to node i.

The problem was modeled with threads to perform a parallel depth-first branch and bound search. For background information on parallel searches, see [6]. The threads were implemented in a work pile paradigm, see Chapter 16 in [5]. The work pile contained equally sized partially expanded branches of the search tree. The threads obtained partially expanded branches from the work pile and fully expanded them in search of a solution. The initial bound of the problem was obtained by a greedy heuristic, see [8]. For testing purposes, the heuristic always returned the optimal solution. Therefore, it was the task of the threads to verify the optimality of the heuristic. Synchronization was required for accessing the work pile and for solution updates. Yet, recall the previous experiment showed that NT's mutex performed poorly when extensive synchronization was required. This leads one to believe that a critical section should be used for NT. However, after thorough testing, it was found that, synchronization occurred infrequently enough that it could be implemented by using mutexes without any loss in performance as compared to a critical section. We still chose to report our results using a critical section for NT. In the case of Solaris, a mutex with local scope was used. The data, gr17.tsp with n = 17, were obtained from the TSPLIB at Rice University [17]. Table 7 shows how long it took to verify optimality using processes in each system.

Threads	NT		Solaris					
			Bound		CL=4		Unbound	
	Time	σ	Time	σ	Time	σ	Time	σ
1	149.86	0.05	152.05	0.01	152.08	0.04	152.08	0.02
2	74.96	0.01	76.06	0.01	76.06	0.01	76.06	0.02
3	50.02	0.01	50.76	0.01	50.74	0.01	76.29	0.22
4	37.59	0.07	38.20	0.04	38.17	0.04	76.37	0.33
8	37.90	0.26	38.36	0.24	38.15	0.02	76.35	0.34
16	38.17	0.24	38.78	0.21	38.18	0.04	76.97	0.29

Table 7: Comparison of the performance of the TSP using threads to perform a parallel depth-first branch and bound search (Data: gr17.tsp, n = 17).

The NT version of the TSP slightly outperformed the Solaris version. Both systems were able to achieve an almost linear speed up (3.9+). The Solaris benchmarks again showed that when the LWP pool size was four the performance was equivalent to using four bound threads. Another observation was that when using two or more of Solaris's unbound threads the performance was equal to using two of Solaris's bound threads. This would suggest that the thread library used two LWPs although two LWPs were not requested. This is the only experiment where Solaris's thread library changed the size of the LWP pool.

3.6 Threads With CPU Bursts

The last experiment determined how each operating system's thread implementation would perform on processes that had many threads with CPU bursts. This is analogous to applications that involve any type of input and output (I/O), e.g., networking or client/server applications, such as back end processing on a SQL server. The experiment was performed by executing a process that created many threads. Each thread would repeatedly be idle for one second and then require the CPU for a variable number of seconds. Three burst lengths were explored, one less than the idle time (0.5 sec.), one equal to the idle time (1.0 sec.) and one greater than the idle time (4 sec.). The amount of required CPU time causes the threads to act as if they are I/O-bound, equally-bound, or CPU- bound, respectively. Tables 8 – 10 show how long it took to complete the processes in each system.

Threads	NT		Solaris					
			Bound		CL=4		Unbound	
	Time	σ	Time	σ	Time	σ	Time	σ
1	7.58	0.01	7.53	0.00	4.99	0.04	4.96	0.13
4	7.58	0.01	7.58	0.05	4.97	0.15	5.93	0.12
8	9.17	0.19	8.94	0.10	7.23	0.28	10.82	0.23
16	12.25	0.30	12.90	0.14	10.97	0.24	20.88	0.16
32	21.47	0.03	21.54	0.11	20.93	0.10	41.11	0.11
64	41.70	0.02	41.80	0.05	40.97	0.50	81.57	0.11

Table 8: Comparison of the performance of processes with threads that require the CPU for intervals that are less than their idle time (I/O-Bound).

Threads	NT		Solaris					
			Bound		CL=4		Unbound	
	Time	σ	Time	σ	Time	σ	Time	σ
1	10.08	0.01	10.05	0.02	9.96	0.13	9.94	0.21
4	10.08	0.01	10.13	0.08	9.95	0.12	10.99	0.11
8	14.17	0.18	13.24	0.16	11.11	0.35	20.92	0.05
16	22.27	0.06	22.11	0.25	20.95	0.11	40.99	0.04
32	41.92	0.05	41.58	0.06	41.09	0.28	81.15	0.13
64	81.82	0.03	82.13	0.12	81.81	0.36	162.39	0.20

Table 9: Comparison of the performance of processes with threads that require the CPU for intervals that are equal to their idle time (Equally-Bound).

Threads	NT		Solaris					
			Bound		CL=4		Unbound	
	Time	σ	Time	σ	Time	σ	Time	σ
1	25.15	0.02	25.10	0.01	24.98	0.07	25.00	0.00
4	25.16	0.01	25.33	0.17	24.98	0.08	40.99	0.08
8	43.36	0.25	42.50	0.23	42.06	0.60	81.04	0.11
16	82.25	0.06	82.25	0.31	81.78	0.48	108.84	0.27
32	162.25	0.04	162.63	0.14	162.68	0.39	237.67	0.36
64	322.64	0.01	323.67	0.31	323.81	0.31	431.44	0.44

Table 10: Comparison of the performance of processes with threads that require the CPU for intervals that are greater than their idle time (CPU-Bound).

The experiments showed a few differences in the performance between Solaris's bound threads, Solaris's threads with restricted concurrency and NT's threads. A noticeable difference in performance occurred in the first two cases, shown in Tables 8 and 9, where the threads required the CPU for intervals that were less than or equal to their idle time. In these cases, the Solaris version using restricted concurrency showed a slightly better performance in comparison to NT's threads or Solaris bound and unbound threads. This can be directly attributed to Solaris's two-tier system. In this case, it was shown that optimal performance could be achieved by setting the concurrency level to the number of CPUs and creating as many unbound threads as needed. This logically creates one LWP for each CPU. Recall the operating system is only aware of the LWPs. This coupled with the FIFO scheduling of Solaris's thread library keeps its bookkeeping to a minimal while maximizing the concurrency.

There were also notable differences in performance in the last case, Table 10, where the CPU intervals were greater than the idle time, CPU-bound. The results of Solaris's bound threads and NT's threads were similar to the fourth experiment, Section 3.3, Table 4. NT's threads out performed Solaris's bound threads as the number of threads increased.

4. Conclusions

Both Windows NT and Solaris were able to utilize multiprocessors. Their performance scaled well with the number of CPUs. However, there is a lack of documentation pertaining to the performance issues of each system. Microsoft and Sun have taken steps in the right direction with the availability of documentation at their respective web sites [7] and [18]. However, little is written on the performance impact of each design. Yet, we found that each implementation can have significant performance implications on various types of applications.

The experiments showed that Windows NT's thread implementation excelled at CPU intensive tasks that had limited synchronization or only intra-process synchronization. Therefore, NT's threads can be greatly exploited on applications such as parallel computations or parallel searches. The experiments also showed that NT's mutex performed poorly compared to Solaris's mutex, when extensive synchronization was required. However, NT's critical section provided significantly better performance than Solaris's mutex. Therefore, for NT, a critical section should be used to implement extensive intra-process synchronization. Another NT observation was that to achieve optimal performance the number of threads used by a process for the execution of a parallel search or computation should be approximately equal to the number of CPUs. Although, it was found that the exact number of threads was dependent on the specific problem, its implementation and the specific data set being used, also see [6]. It is also worth noting, that both systems grew erratic as the number of executing CPU intensive threads grew larger than the number of processors. This was especially true in the NT case. Responsiveness was sluggish on heavily loaded systems and often required dedicated system usage.

Solaris's thread API proved to be more flexible, at the cost of complexity. We found that the exploitation of multiprocessors required a thorough understanding of the underlying OS architecture. However, we also found Solaris's two-tier design to have a positive performance impact on tasks with bursty processor requirements. This suggests that Solaris threads are well suited for applications such back end processing or client/server applications, where a server can create threads to respond to a client's requests. In addition, we found the Solaris thread library's automatic LWP pool size control to be insignificant. We found in most cases, the programmer can achieve desirable performance with unbound threads and a restricted concurrency level that is equal to the number of processors.

In conclusion, the advent of relatively inexpensive multiprocessor machines has placed a critical importance on the design of mainstream operating systems and their implementations of threads. Threads have become important and powerful indispensable programming tools. They give programmers the ability to execute tasks concurrently. When used properly they can dramatically increase performance, even on a uniprocessor machine. However, threads are new to mainstream computing and are at a relatively early stage of development. Thus, arguments exist on how threads should be implemented. Yet, one should remember that differences between implementations are simply tradeoffs. Implementers are constantly trying to balance their implementations by providing facilities they deem the most important at some acceptable cost.

Note there has been a movement to standardize threads. IEEE has defined a thread standard POSIX 1003.1c-1995 that is an extension to the 1003.1 Portable Operating System Interface (POSIX). The standard, called *pthreads,* is a library-based thread API. It allows one to develop thread applications cross platform. However, IEEE does not actually implement the library. It only defines what should be done, the API. This leaves the actual implementation up to the operating system developer. Usually the pthreads library is built on the developer's own thread implementation. It is simply a wrapper over the developers' own implementation and thus, all features may or may not exist. In the case where the OS does not have a thread implementation, the library is solely user based, and thus can not exploit multiprocessors.

5. Acknowledgements

This research is supported in part by ONR grant N00014-96-1-1057.

References

[1] Berg, D.J.; Lewis, B.: *Threads Primer: A Guide to Multithreaded Programming*, SunSoft Press, 1996

[2] Collins, R.R.: Benchmarks: Fact, Fiction, or Fantasy, *Dr. Dobb's Journal*, March 1998

[3] El-Rewini, H.; Lewis, T.G.: *Introduction to Parallel Computing*, Prentice Hall, 1992

[4] Intel: The Pentium Pro Processor Performance Brief at : http://www.intel.com/procs/perf/highend/

[5] Kleiman, S.; Shah, D.; Smaalders, B.: *Programming With Threads*, SunSoft Press, 1996

[6] Kumar, V.; Grama, A.; Gupta, A.; Karypis, G.: *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Chapter 8, Benjamin Cummings, 1994

[7] Microsoft: Microsoft Developer Network, http://www.microsoft.com/msdn

[8] McAloon, K.; Tretkoff, C.: *Optimization and Computational Logic*, Wiley, 1996

[9] Prasad, S.: Weaving a Thread, *Byte,* October 1996

[10] Richter, J.: *Advanced Windows NT: The Developer's Guide to the Win32 Application Programming Interface*, Microsoft Press, 1994

[11] SunSoft: Multithreaded Implementations and Comparisons, A White Paper, 1996

[12] SunSoft: Multithread Programming Guide, A User Manual, 1994

[13] SunSoft: Solaris SunOS 5.0 Multithread Architecture, A White Paper, 1991

[14] Tanenbaum, A.S.: *Distributed Operating Systems*, Prentice Hall, 1995

[15] Tomlinson, P.: Understanding NT: Multithreaded Programming, *Windows Developer's Journal*, April 1996

[16] Thompson, T.: The World's Fastest Computers, *Byte*, January 1996

[17] TSPLIB: http://softlib.rice.edu/softlib/tsplib/

[18] http://www.sun.com/workshop/threads/

A System for Structured High-Performance Multithreaded Programming in Windows NT

John Thornley, K. Mani Chandy, and Hiroshi Ishii
Computer Science Department
California Institute of Technology
Pasadena, CA 91125, U.S.A.
{john-t, mani, hishii}@cs.caltech.edu

Abstract

With the advent of inexpensive multiprocessor PCs, multithreading is poised to play an important role in computationally intensive business and personal computing applications, as well as in science and engineering. However, the difficulty of multithreaded programming remains a major obstacle. Windows NT support for threads is well suited to systems programming, but is too unstructured when multithreading is used for the purpose of speeding up program execution. In this paper, we describe a system for structured multithreaded programming. Thread creation operations are multithreaded variants of blocks and loops, and synchronization objects are based on Boolean flags and integer counters. With this system, most multithreaded program development can be performed using traditional sequential methods and tools. The system is integrated with Windows NT and Microsoft Developer Studio Visual C++. We are developing a variety of applications in collaboration with other researchers, to demonstrate the power of structured multithreaded programming on commodity multiprocessors running Windows NT. In one benchmark application (aircraft route optimization), we achieved better performance on a quad-processor Pentium Pro system than the best results reported on expensive supercomputers.

1. Introduction

In the past, high-performance multithreading has been synonymous with either scientific supercomputing, real-time control, or achieving high throughput on multi-user servers. The idea of dividing a computationally intensive program into multiple concurrent threads to speed up execution on multiprocessor computers is well established. However, this kind of high-performance multithreading has made very little impact in mainstream business and personal computing, or even in most areas of science and engineering. Part of the reason has been the rarity and high cost of multiprocessor computer systems. Another part of the reason is the difficulty of developing multithreaded programs, as compared to equivalent sequential programs.

The recent advent of inexpensive multiprocessor PCs and commodity OS support for lightweight threads opens the door to an important role for high-performance multithreading in all areas of computing. Dual-processor and quad-processor PCs are now available for a few thousand dollars. Mass-produced multiprocessors with 8, 16, and more processors will soon follow. Windows NT [2] can already support hundreds of fine-grained threads with low overhead, even on single-processor machines, and future releases will be even more efficient. Examples of applications that could use multithreading to improve performance include spreadsheets, CAD/CAM, three-dimensional rendering, photo/video editing, voice recognition, games, simulation, and resource management.

The biggest obstacle that remains is the difficulty of developing efficient multithreaded programs. General-purpose thread libraries, including the Win32 API [1][8] supported by Windows NT, are well suited to systems programming applications of threads, e.g., control systems, database systems, and distributed systems. However, the interface provided by general-purpose thread libraries is less well suited to applications where threads are used for the purpose of speeding up program execution. General-purpose thread management is unstructured and synchronization operations are complex and error-prone. The unpredictable interactions of multiple threads introduce many problems (e.g., race conditions and deadlock) that do not occur in sequential programming. In many regards, general-purpose thread libraries are the assembly language of high-performance multithreaded programming.

In this paper, we describe our ongoing research to develop a system for structured high-performance multithreaded programming on top of the general-purpose thread support provided by operating systems such as Windows NT. The key attributes of our system are as follows:

- Structured thread creation constructs are based on sequential blocks and `for` loops.
- Structured synchronization constructs are based on Boolean flags and integer counters.
- Subject to a few simple rules, multithreaded execution is deterministic and produces the same results as sequential execution.
- Lock synchronization is provided for nondeterministic algorithms.
- Barrier synchronization is provided for efficiency.

Our system is supported at two levels: (i) Sthreads, a structured thread library, and (ii) Multithreaded C, a set of pragmas transformed into Sthreads calls by a preprocessor. Sthreads is easily implemented as a thin layer on top of general-purpose thread libraries. The Multithreaded C preprocessor is a simple source-to-source transformation tool that involves no complex program analysis. Therefore, the entire system is highly portable.

One of the major strengths of our system is that much of the development of a multithreaded application can be performed using ordinary sequential methods and tools. The Multithreaded C preprocessor is integrated with Microsoft Developer Studio Visual C++ [7]. Applications can either be built either as multithreaded applications (as indicated by the pragmas) or as sequential applications (by ignoring the pragmas). For deterministic applications, most development, testing, and debugging can be performed using the sequential version of the application. Absence of race conditions and deadlock can be verified in the context of sequential execution. Nondeterministic applications can usually be developed with large deterministic components.

The focus of our work is on commodity multiprocessors and operating systems, in particular multiprocessor PCs and Windows NT. However, our programming system is portable across platforms ranging from low-end PCs to high-end workstations and supercomputers. For this reason, the value of our work is not restricted to developers of applications for commodity systems. Our portable system for high-performance multithreaded programming also allows for high-end applications to be developed, tested, and debugged on accessible low-end platforms.

The remainder of this paper is organized as follows: in Section 2, we discuss the interface and performance of Windows NT support for multithreading; in Section 3, we describe our structured multithreaded programming system; in Section 4, we report in some detail on one particular application (aircraft route optimization) that we have developed using our system; in Section 5, we give a brief outline of several other applications that we are developing; in Section 6, we compare our system with related work; and in Section 7, we summarize and conclude.

2. Windows NT Multithreading

In this section, we describe the interface and performance of standard Windows NT thread support. Since our emphasis is on commodity systems and applications, Windows NT is the ideal platform on top of which to build our system for structured multithreaded programming. The following are particularly important to us: (i) the Windows NT thread interface provides the functionality that we require for our system, and (ii) the Windows NT thread implementation efficiently supports large numbers of lightweight threads.

2.1. Windows NT Thread Interface

Windows NT implements the Win32 thread API. This interface provides all the functionality that is needed for high-performance multithreaded programming. However, because the scope of the Win32 thread API is general-purpose, the interface is more complicated and less structured than is desirable when threads are used for the purpose of speeding up program execution. For this reason, we have built a less general, less complicated, and more structured layer on top of the functionality provided by the Win32 thread API.

The Win32 thread API is typical of other general-purpose thread libraries, e.g., Pthreads [6] and Solaris threads [4]. It provides a set of function calls for creating and terminating threads, suspending and resuming threads, synchronizing threads, and controlling the scheduling of threads using priorities. The interface contains a large number of constants and types, a large number of functions, and many optional arguments. Although all these operations have important uses in general-purpose multithreading, only a structured subset of this functionality is required for our purpose.

As with other general-purpose thread libraries, Win32 thread creation is unstructured. A thread is created by passing a function pointer and an argument pointer to a `CreateThread` call. The new thread executes the given function with the given argument. The thread can be created either runnable or suspended. After creation, there is no special relationship or synchronization between the created thread and the creating thread. For example, the created thread may outlive the creating thread, causing problems if the created thread references variables in the creating thread. Many unstructured operations are permitted on threads. For example, one thread can arbitrarily suspend, resume, or terminate the execution of another thread.

The Win32 thread API provides a large range of synchronization operations. A thread can synchronize on the termination of another thread, or on the termination of one or all of a group of other threads. Critical section, mutex, semaphore, and event objects, and interlocked operations allow many other forms of synchronization between threads. There are many options associated with these synchronization operations, particularly with operations on event objects. All of these synchronization operations almost inevitably introduce nondeterminacy to a program. Nondeterminacy is an implicit part of most systems programming applications of threads, but is best avoided if possible in other applications, because of the difficulty it adds to testing, debugging, and performance prediction.

To summarize, Windows NT supports a rich but complex set of general-purpose thread management and synchronization operations. We implement a simpler layer for structured multithreaded programming on top of the Windows NT thread interface.

2.2. Windows NT Thread Performance

Lightweight multithreading is an integral part of our programming model. If multithreaded applications are to make an impact in commodity software, they must be able to execute efficiently on systems with differing numbers of processors, and dynamically adapt to varying background load conditions. The best way to achieve this is to build applications with large numbers of dynamically created lightweight threads that can take advantage of whatever processing resources become available during execution. The traditional scientific supercomputing model of one static thread per processor on an otherwise unloaded system is not sufficient in the commodity software domain.

We have performed experiments that demonstrate that Windows NT supports large numbers of lightweight threads efficiently. Specifically, on single-processor, dual-processor, and quad-processor Pentium Pro systems running Windows NT, we have demonstrated the following:

- Thread creation and termination overheads are low.
- Synchronization overheads are low.
- Hundreds of threads can be supported without significant performance degradation.
- On single-processor systems, multithreaded programs with many threads can run as fast as equivalent sequential programs.
- On multiprocessor systems, multithreaded programs with many threads can achieve good speedups over equivalent sequential programs.

We have developed many small and large programs which demonstrate that multithreaded Windows NT applications can execute efficiently on both single-processor and multiprocessor systems, without any kind of reconfiguration. We have found that good speedups are maintained with much larger numbers of threads than processors.

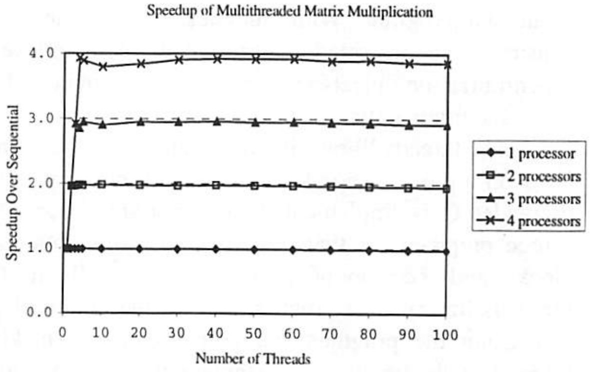


Figure 1: Speedup of multithreaded matrix multiplication over sequential matrix multiplication on one to four processors of a quad-processor Pentium Pro system.

As a simple example, Figure 1 shows the speedup of multithreaded matrix multiplication over sequential matrix multiplication for 1000-by-1000 element matrices on a 200 MHz quad-processor Pentium Pro system running Windows NT Server 4.0. A straightforward three nested-loops algorithm is used. Sequential matrix multiplication takes 50 seconds. In the multithreaded version, the iterations in the outer loop are evenly partitioned among the threads. (The intention of this example is to demonstrate Windows NT multithreaded performance characteristics, not optimal matrix multiplication algorithms.) Near-perfect speedups are achieved and are maintained with large numbers of threads.

To summarize, we have found that Windows NT efficiently supports large numbers of lightweight threads. Our structured multithreaded programming system is implemented as a very thin layer on top of Windows NT support for threads.

3. A System for Structured Multithreaded Programming

In this section, we describe the features of our structured multithreaded programming system. Since it is beyond the scope of this short paper to describe all the details of our system, we aim to give an overview of the fundamental constructs and development methods.

3.1. Overview

We are developing a two-level approach to structured multithreaded programming in ANSI C [3]. The higher level is a pragma-based notation (Multithreaded C), and the lower level is a structured thread library (Sthreads). In both Multithreaded C and Sthreads, thread creation constructs are multithreaded variants of sequential block and for loop constructs. With Multithreaded C, the constructs are supported as pragma annotations to a sequential program. With Sthreads, exactly the same constructs are supported as library calls. At both levels, synchronization objects and operations are supported as Sthreads library calls.

The Sthreads library is implemented as a very thin layer on top of the Windows NT thread interface. Multithreaded C is implemented as a portable source-to-source preprocessor that directly transforms annotated blocks and for loops into equivalent calls to the Sthreads library. The programmer has the option of either using the pragmas and preprocessor or making Sthreads calls directly. The Sthreads library and Multithreaded C preprocessor are integrated with Microsoft Developer Studio Visual C++. Building a project automatically invokes the preprocessor where necessary and links with the Sthreads library.

3.2. Multithreadable Blocks and for Loops

Multithreadable blocks and for loops are ordinary blocks and for loops for which multithreaded execution is equivalent to sequential execution. In Multithreaded C, multithreadable blocks and for loops are expressed using the multithreadable pragma. It is obvious that the pragma can be applied blocks and for loops in which the statements or iterations are independent of each other. As a simple example, consider the following program to sum the elements of a two-dimensional array:

```
void SumElements(
    float A[N][N], float *sum, int numThreads)
{
    int i;
    float rowSum[N];

    #pragma multithreadable \
            mapping(blocked(numThreads))
    for (i = 0; i < N; i++) {
        int j;
        rowSum[i] = 0.0;
        for (j = 0; j < N; j++)
           rowSum[i] = rowSum[i] + A[i][j];
    }
    *sum = 0.0;
    for (i = 0; i < N; i++)
        *sum = *sum + rowSum[i];
}
```

Multithreaded execution of the for loop is equivalent to sequential execution because the iterations all modify different rowSum[i] and j variables. The arguments following the pragma indicate that multithreaded execution should assign iterations to numThreads different threads using a blocked mapping. There is a rich set of options that control the mapping of iterations to threads.

3.3. Synchronization Using Flags and Counters

Flags and counters are provided to express deterministic synchronization within multithreadable blocks and for loops. Flags support Set and Check operations. Initially, a flag is not set. A Set operation on a flag atomically sets the flag. A Check operation on a flag suspends until the flag is set. Once a flag is set, it remains set. Counters support Increment and Check operations. A counter has an initial value of zero. An Increment operation on a counter atomically increments the value of the counter. A Check operation on a counter suspends until the value of the counter reaches a given value. The value of a counter cannot be decremented. The only way to test the value of a flag or counter is with a Check operation. As a simple example, consider the following program to sum the elements of a two-dimensional array:

```
void SumElements(
    float A[N][N], float *sum, int numThreads)
{
    int i;
    SthreadCounter counter;

    SthreadCounterInitialize(&counter);
    #pragma multithreadable \
            mapping(blocked(numThreads))
    for (i = 0; i < N; i++) {
        int j;
        float rowSum;
        rowSum = 0.0;
        for (j = 0; j < N; j++)
           rowSum = rowSum + A[i][j];
        SthreadCounterCheck(&counter, i);
        *sum = *sum + rowSum;
        SthreadCounterIncrement(&counter, 1);
    }
    SthreadCounterFinalize(&counter);
}
```

Without the counter operations, multithreaded execution of the for loop would not be equivalent to sequential execution, because the iterations all modify the same *sum variable. However, the counter operations ensure that multithreaded execution is equivalent to sequential execution. In sequential execution, the iterations are executed in increasing order and the SthreadCounterCheck operations succeed without suspending. In multithreaded execution, the counter operations ensure that the operations on *sum occur atomically and in the same order as in sequential execution. Iteration i suspends at the SthreadCounterCheck operation until iteration i - 1 has executed the SthreadCounterIncrement operation.

3.4. Sequential Development Methods

The equivalence of multithreaded and sequential execution of multithreadable blocks and `for` loops allows multithreaded programs to be developed using ordinary sequential methods and tools. The `multithreadable` pragma is an assertion by the programmer that the block or `for` loop *can* be executed in a multithreaded manner without changing the results of the program. It is not a directive that the block or `for` loop *must* be executed in a multithreaded manner. The Multithreaded C preprocessor has two modes: sequential mode in which the `multithreadable` pragma is ignored and multithreaded mode in which the `multithreadable` pragma is transformed into Sthreads calls. Programs can be developed, tested, and debugged in sequential mode, then executed in multithreaded mode for performance. In addition, performance analysis and tuning can often be performed in sequential mode.

The advantages of this approach to multithreaded programming are clear. However, the programmer is responsible for correct use of the `multithreadable` pragma. Fortunately, the rules for correct use of the pragma are straightforward and can be stated entirely in terms of sequential execution. In sequential execution, accesses to shared variables must be separated by `Set` and `Check` operations or `Increment` and `Check` operations, and the `Check` operations must not suspend. In multithreaded execution, the synchronization operations will ensure that accesses to shared variables occur atomically and in the same order as in sequential execution. Therefore, multithreaded execution will be equivalent to sequential execution.

3.5. Determinacy

Determinacy of results is an important consequence of the equivalence of multithreaded and sequential execution. If sequential execution is deterministic (which is usually the case), multithreaded execution will also be deterministic. Determinacy is usually desirable, since program development and debugging can be difficult when different runs produce different results. In many other multithreaded programming systems, determinacy is difficult to ensure. For example, locks, semaphores, and many-to-one message passing almost always introduce race conditions and hence nondeterminacy. However, nondeterminacy is important for efficiency in some algorithms, e.g., branch-and-bound algorithms.

3.6. Multithreaded Blocks and `for` Loops

Multithreaded blocks and `for` loops are blocks and `for` loops that *must* be executed in a multithreaded manner. Multithreaded execution is not necessarily equivalent to sequential execution. In Multithreaded C, multithreaded blocks and `for` loops are expressed using the `multithreaded` pragma. Unlike the `multithreadable` pragma, the `multithreaded` pragma is transformed into Sthreads calls by the Multithreaded C preprocessor in both sequential and multithreaded mode.

3.7. Synchronization Using Locks

Locks are provided to express nondeterministic synchronization, usually mutual exclusion, within multithreaded blocks and `for` loops. Our locks support the usual `Acquire` and `Release` operations. The order in which concurrent `Acquire` operations succeed is nondeterministic. Therefore, there is very little use for locks within multithreadable blocks and `for` loops. As a simple example, consider the following program to sum the elements of a two-dimensional array:

```
void SumElements(
    float A[N][N], float *sum, int numThreads)
{
    int i;
    SthreadLock lock;

    SthreadLockInitialize(&lock);
    #pragma multithreaded \
            mapping(blocked(numThreads))
    for (i = 0; i < N; i++) {
        int j;
        float rowSum;
        rowSum = 0.0;
        for (j = 0; j < N; j++)
           rowSum = rowSum + A[i][j];
        SthreadLockAcquire(&lock);
        *sum = *sum + rowSum;
        SthreadLockRelease(&lock);
    }
    SthreadLockFinalize(&lock);
}
```

Like the flag operations in the program in Section 3.3, the lock operations in this program ensure that the operations on `*sum` occur atomically. However, unlike the flag operations, the lock operations do not ensure that the operations on `*sum` occur in the same order as in sequential execution, or even in the same order each time the program is executed. Therefore, since floating-point addition is not associative, the program may produce different results each time it is executed. However, because execution order is less restricted, this program allows more concurrency than the program in Section 3.3. This is an example of the commonly occurring tradeoff between determinacy and efficiency.

3.8. Synchronization Using Barriers

Barriers are provided to express collective synchronization of a group of threads in cases when thread termination and recreation is too expensive. Our barriers support the usual `Pass` operation. All the threads in a group must enter the `Pass` operation before all the threads in the group are allowed to leave the `Pass` operation. In current systems, the cost of N threads executing a `Pass` operation is less than the cost of creating and terminating N threads. Therefore, a typical use of barriers is to replace a sequence of multithreadable loops with a single multithreaded loop containing a sequence of barrier `Pass` operations. However, with modern lightweight thread systems such as Windows NT, we are discovering that barriers are required for efficiency in very few circumstances.

3.9. The Sthreads Interface

The examples that we have given so far are expressed using the Multithreaded C pragma notation. As we described previously, there is a direct correspondence between the pragma notation for thread creation and the Sthreads library functions that support thread creation. As a simple example, the following is program from Section 3.3 implemented using Sthreads:

```
typedef struct {
    float (*A)[N];
    float *sum;
    SthreadCounter *counter;
} LoopArgs;

void LoopBody(
    int i, int notused1, int notused2,
    LoopArgs *args)
{
    int j;
    float rowSum;
    rowSum = 0.0;
    for (j = 0; j < N; j++)
       rowSum = rowSum + (args->A)[i][j];
    SthreadCounterCheck(args->counter, i);
    *(args->sum) = *(args->sum) + rowSum;
    SthreadCounterIncrement(args->counter, 1);
}

void SumElements(
    float A[N][N], float *sum, int numThreads)
{
    int i;
    SthreadCounter counter;
    LoopArgs args;

    SthreadCounterInitialize(&counter);
    args.A = A;
    args.sum = sum;
    args.counter = &counter;
    SthreadRegularForLoop(
        (void (*)(int, int, int, void *))
        LoopBody,
        (void *) &LoopArgs,
        0, STHREAD_CONDITION_LT, N, 1,
        1, STHREAD_MAPPING_BLOCKED,
        numThreads,
        STHREAD_PRIORITY_PARENT,
        STHREAD_STACK_SIZE_DEFAULT);
    SthreadCounterFinalize(&counter);
}
```

Although this program is syntactically more complicated than the Multithreaded C version, it is considerably less complicated than the same program expressed using Windows NT threads. The mechanics of creating threads, assigning iterations to threads, and waiting for thread termination is handled within the Sthreads library call.

3.10. Performance Issues

In our multithreaded programming system, obtaining good performance is under the control of the programmer. The following issues must be taken into account:

- Multithreading overheads: Threading and synchronization operations are time consuming.
- Load balancing: Enough concurrency must be expressed to keep the processors busy.
- Memory contention: Locality in memory access patterns prevents memory contention.

The key tradeoff is granularity. If multithreading is too fine-grained, the overheads will swamp the useful computation. If multithreading is too coarse grained, the processors will spend too much time idle. Fortunately, Windows NT supports lightweight threads with low overheads.

3.11. Implementation and Performance on Windows NT

Sthreads for Windows NT is implemented as a very thin layer on top of the Win32 thread API. As a consequence, there is essentially no performance overhead associated with using Sthreads or Multithreadable C, as compared to using Win32 threads directly.

Multithreadable blocks and `for` loops are implemented as a sequence of `CreateThread` calls followed by a `WaitForSingleObject` call on an event. Terminating threads perform an `InterlockedDecrement` call on a counter, and the last thread to terminate performs a `SetEvent` call on the event. Flags are implemented directly as Win32 events. Counters are implemented as linked lists of Win32 events, with one event for every value on which some thread is waiting. Locks are implemented directly as

Win32 critical sections. Barriers are implemented as a pair of Win32 events and a Win32 critical section.

On a single-processor 200 MHz Intel Pentium Pro system running Windows NT Server 4.0, the time to create and terminate each thread in a multithreadable block or `for` loop is approximately 500 microseconds. The time to initialize and finalize a flag, counter, lock, or barrier is between 5 and 20 microseconds. The time to perform a synchronization operation on a flag, counter, lock, or barrier is less than 10 microseconds.

3.12. Current Status

The Sthreads implementation for Windows NT is complete and robust. Over the last year, we have developed many different multithreaded applications using Sthreads on quad-processor Pentium Pro systems running Windows NT. This academic year, we taught an undergraduate class at Caltech on multithreaded programming, with Sthreads used for all the homework assignments. We are in the process of implementing Sthreads on top of Pthreads for a variety of Unix platforms, including Sun UltraSPARC, SGI Origin, and HP Exemplar. We have developed a comprehensive, platform-independent test suite for Sthreads and a timing suite to compare the performance of different Sthreads implementations.

At the time of writing, the Multithreaded C preprocessor is still under development. We hope to make a prototype preprocessor available by the fourth quarter of 1998. In the meanwhile, we are developing multithreaded applications by making Sthreads calls directly. Because of the direct correspondence between the pragma annotations and Sthreads calls, the design and development of algorithms and programs is the same in Sthreads and Multithreaded C. The only difference is in the syntax.

4. An Example Application: Aircraft Route Optimization

In this section, we report on one example application that we have developed. The Aircraft Route Optimization Problem is part of the U.S. Air Force Rome Laboratory C3I Parallel Benchmark Suite [5]. For this application, we achieved better performance using Sthreads on a quad-processor Pentium Pro system running Windows NT than the best reported results for message-passing programs running on expensive Cray and SGI supercomputers with up to 64 processors. The flexibility of shared-memory, lightweight multithreading, and sequential development methods allowed us to develop a much more sophisticated and efficient algorithm than would be possible on a message-passing supercomputer.

4.1. The C3I Parallel Benchmark Suite

The U.S. Air Force Rome Laboratory C3I Parallel Benchmark Suite consists of eight problems chosen to represent the essential elements of real C3I (Command, Control, Communication, and Intelligence) applications. Each problem consists of the following:

- A problem description giving the inputs and required outputs.
- An efficient sequential program (written in C) to solve the problem.
- The benchmark input data.
- A correctness test for the benchmark output data.

For some of the problems, a parallel message-passing program is also provided. Rome Laboratory maintains a publicly accessible database of reported performance results.

The C3I Parallel Benchmark Suite provides a good framework for evaluating our structured multithreaded programming system. The problems are computationally intensive and involve a variety of complex algorithms and data structures. The sequential program provides us with a good starting point and a fair basis for performance comparison. The performance database allows us to compare our results with those of other researchers. For these reasons, we are developing multithreaded solutions to several of the C3I Parallel Benchmark Suite problems.

4.2. The Aircraft Route Optimization Problem

The task in the Aircraft Route Optimization Problem is to find the lowest-risk path for an aircraft from an origin point to a set of destination points in the airspace over an uneven terrain. The risk associated with each transition in the airspace is determined by its proximity to a set of threats. The problem involves realistic constraints on aircraft speed and maneuverability. The aircraft is also constrained to fly above the underlying terrain and beneath a given ceiling altitude.

The problem is essentially the single-source, multiple-destination shortest path problem with a large, sparsely connected graph. The airspace for the benchmark is 100 km by 100 km in area and 10 km in altitude, discretized at 1 km intervals. The 100,000 positions in space correspond to 2,600,000 nodes in the graph, since each position can be reached from 26 different directions. Because of aircraft speed and maneuverability constraints, each node is connected to only nine or ten geographically adjacent nodes. Therefore, the graph consists of approximately 2.6 million nodes and 26 million edges.

4.3. Sequential Algorithm

The sequential algorithm to solve the Aircraft Route Optimization Problem is based on a queue of nodes. Initially the queue is empty except for the origin node. At each step, one node is removed from the queue. Valid transitions from this source node to all adjacent destination nodes are considered. For each destination node, if the path to the node via the source node is shorter than the current shortest path to the node, the path to the node is updated and the node added to the queue. The algorithm continues until the queue is empty, at which stage the shortest paths to all reachable nodes have been computed.

The queue is ordered on path length so that shorter paths are expanded before longer paths. This has a significant effect on performance. Without ordering, longer paths are expanded, then discarded when shorter paths to the same points are expanded later in the computation. However, whether the queue is ordered, partially ordered, or unordered does not affect the results of the algorithm.

4.4. Multithreaded Algorithm

The most straightforward approach to obtaining parallelism in the Aircraft Route Optimization Problem is to geographically partition the airspace into blocks, with one thread (or process) responsible for each block. Each thread runs the sequential algorithm on its own block using its own local queue and periodically exchanges boundary values with neighboring blocks. This approach is particularly appealing on distributed-memory, message-passing platforms, because memory can be permanently distributed according to the blocking pattern. If the threads execute a reasonably large number of iterations between boundary exchanges, good load balance can be achieved.

The problem with this algorithm is that, as the number of blocks/threads is increased the total amount of computation also increases. Therefore, any speedup is based on an increasingly inefficient underlying algorithm. At any time, the local queues in most blocks contain paths that are too long and are irrelevant to the actual shortest paths. The processors are kept busy performing computation that is later discarded. At any given time, it is only productive to work on an irregular and unpredictable subset of the graph. However, irregular and adaptive blocking schemes do not solve the problem, since there is usually equal work available in all blocks. The issue is the distinction between productive and unproductive work.

Our solution is to statically partition the airspace into a large number of blocks and to use a much smaller number of threads. A measure of the average path length is maintained with each local queue. At each step, the blocks with local queues containing the shortest paths are assigned to the threads. Therefore, the subset of blocks that are active and the assignment of blocks to threads change dynamically throughout program execution. This algorithm takes advantage of the symmetric multiprocessing model, in which all threads can access the entire memory space with uniform cost. It also takes advantage of the lightweight multithreading model to achieve good load balance, since the workload within each thread at each step is highly variable.

The ability to develop, test, and debug using sequential methods was crucial in the development of this sophisticated multithreaded algorithm. The entire program was tested and debugged in sequential mode before multithreaded execution was attempted. In particular, development of the complex boundary exchange and queue update algorithms would have been considerably more difficult in multithreaded mode.

The ability to analyze and tune performance using sequential methods was also very important. Good performance depended on exposing enough parallelism without significantly increasing the total amount of computation. We determined efficient values for the number of blocks, the number of threads, and the number of iterations between boundary exchanges by measuring computation times and operation counts of the multithreaded program in running in sequential mode. This detailed analysis would have been very difficult to perform in multithreaded mode. We avoided memory contention in multithreaded mode by avoiding cache misses in sequential mode. The analysis of memory access patterns in sequential mode is much simpler than in multithreaded mode.

4.5. Performance

Other researchers have developed and reported results for message-passing solutions to the Aircraft Route Optimization Problem running on Cray T3D and SGI Power Challenge supercomputers with 16 and 64 processors. We developed our program using Sthreads on a quad-processor 200 MHz Pentium Pro system running Windows NT Server 4.0. (One Pentium Pro processor is approximately the same speed as one SGI Power Challenge processor and twice the speed of one Cray T3D processor.) As shown in Figure 2, our results are better than the results reported on the expensive supercomputers. The reason is the combination of the shared-memory model supported by the Pentium Pro, the lightweight multithreading model supported by Windows NT, and the structured multithreaded programming system with sequential development methods supported by Sthreads.

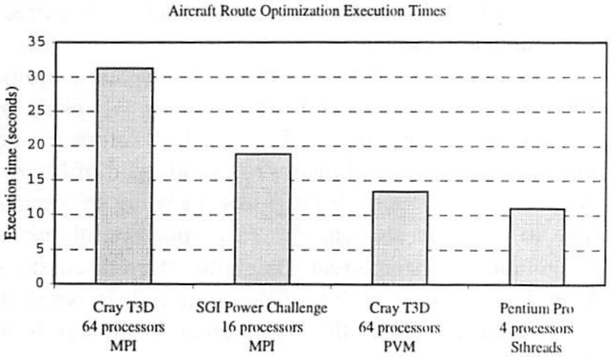


Figure 2: Sthreads on a quad-processor Pentium Pro outperforms message-passing on supercomputers for the Aircraft Route Optimization Problem.

These results should not be misinterpreted as meaning that commodity multiprocessors are now as powerful as supercomputers, or that structured multithreading can obtain super-linear speedups from a small number of processors. For the Aircraft Route Optimization Problem, we obtain approximately three-fold speedup over sequential execution on four processors. However, unlike the message-passing approach, the speedups are obtained from a multithreaded algorithm that adds no significant overheads to the sequential algorithm.

This example is intended as an interesting case study that highlights the strengths of shared-memory, lightweight threads, and structured multithreaded programming. Clearly, there are many other applications for which message-passing supercomputers would outperform a quad-processor Pentium Pro system. Nonetheless, it is significant that inexpensive commodity multiprocessors are now a practical consideration in many areas that were previously the sole domain of supercomputers.

5. Additional Applications under Development

We are in the process of developing a number of other multithreaded applications to demonstrate the power of structured multithreaded programming on commodity multiprocessors running Windows NT. These applications include the following:

- **Volume Rendering:** Volume rendering computes an image of a three-dimensional data volume. The image may or may not be realistic, depending on the nature of the data. We obtain better than 3.5-fold speedups on quad-processor Pentium Pro systems.
- **Terrain Masking:** Terrain masking computes the altitudes at which an aircraft will be visible to ground-based radar systems in an uneven terrain. This problem has obvious applications in military and civil aviation. Terrain masking is part of the C3I Parallel Benchmark Suite. We obtain 3-fold speedups on quad-processor Pentium Pro systems.
- **Threat Analysis:** Threat analysis is a military application that performs a time-stepped simulation of the trajectories of incoming threats and analyses options for intercepting the threats. Threat analysis is part of the C3I Parallel Benchmark Suite. We obtain almost 4-fold speedups on quad-processor Pentium Pro systems.
- **Protein Folding:** Protein folding takes a known protein chain and computes the most likely three-dimensional structures for the molecule. This problem is of vital interest to biochemists involved in drug design. We obtain 3-fold speedups on quad-processor Pentium Pro systems.
- **Molecular Dynamics Simulation:** We are developing a multithreaded implementation of an existing molecular dynamics simulation program (MPSim) that uses the cell multipole method. MPSim consists of more than 100 source files and over 100,000 lines of code. We obtain better than 3.5-fold speedups on quad-processor Pentium Pro systems for simulations involving up to half a million atoms.

All of these applications are extremely computationally intensive. Depending on the input data, the problems may require many hours or days to solve on fast single-processor machines. For this reason, much of the previous work on these problems has been with expensive supercomputers. The computational challenge associated with these problems is not about to disappear with the next generation of fast processors.

6. Comparison with Related Work

Over the years, hundreds of different systems for multithreaded programming have been proposed and implemented. These systems include thread libraries, concurrent languages, sequential languages with pragmas, data parallel languages, automatic parallelizing compilers, and parallel dataflow languages. We have attempted to combine the best attributes of these other approaches into one simple, timely, and highly accessible multithreaded programming system.

The contribution of our system is not in the individual constructs, but in their combination and context. Multithreaded block and `for` loop constructs date back to the 1960s and have been incorporated in many forms in concurrent languages, pragma-based systems, and data parallel languages. Synchronization flags and counters are derived from concepts originally explored in the context of parallel dataflow languages. Locks and barriers are standard synchronization constructs in many thread libraries and concurrent languages.

An important difference between our system and others is the combination of the following: (i) the emphasis on practical development of sophisticated multithreaded programs using sequential methods, (ii) the ability to ensure deterministic execution, but express nondeterminacy where necessary, and (iii) the minimalist integration of these ideas with popular languages, operating systems, and tools. We believe that this is the right approach to making multithreading practical for a wide range of applications running on modern multiprocessor platforms.

7. Conclusion

We are developing a system for structured high-performance multithreaded programming on top of the support that Windows NT provides for lightweight multithreading. Our system consists of a structured thread library and a preprocessor integrated with Microsoft Developer Studio Visual C++. An important attribute of our system is the ability to develop multithreaded programs using traditional sequential methods and tools. We have developed multithreaded applications in a wide range of areas such as optimization, graphics, simulation, and applied science. The performance of these applications on multiprocessor Pentium Pro systems has been excellent. Our experience developing these applications has convinced us that high-performance multithreading is ready to enter the mainstream of computing.

Availability

Information on obtaining the multithreaded programming system described in this paper can be found at `http://threads.cs.caltech.edu/` or can be requested from `threads@cs.caltech.edu`.

Acknowledgments

The applications described in this paper are being developed by the following Caltech students: Eric Bogs, Marrq Ellenbecker, Yuanshan Guo, Maria Hui, Lei Jin, Autumn Looijen, Scott Mandelsohn, Bradley Marker, Jason Roth, Sean Suchter, and Louis Thomas. We are developing the applications in collaboration with the following Caltech faculty, staff, and students: Prof. Peter Schröder in the Computer Science Department, Dr. Jerry Solomon and David Liney in the Computational Biology Department, and Prof. William Goddard III, Dr. Tahir Cagin, and Hao Li in the Materials and Process Simulation Center.

Thanks to Intel Corporation for donating multiprocessor Pentium Pro computer systems. Thanks also to Microsoft Corporation for donating software and documentation. Special thanks to David Ladd of Microsoft University Research Programs for being so responsive to our requests. The design of our multithreaded programming system greatly benefited from discussions with Tim Mattson of Intel Corporation. This research was funded in part by the NSF Center for Research on Parallel Computation under Cooperative Agreement No. CCR-9120008 and by the Army Research Laboratory under Contract No. DAHC94-96-C-0010.

References

[1] Jim Beveridge and Robert Wiener. *Multithreading Applications in Win 32.* Addison-Wesley, Reading, Massachusetts, 1997.

[2] Helen Custer. *Inside Windows NT.* Microsoft Press, Redmond, Washington, 1993.

[3] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[4] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads.* Prentice Hall, Upper Saddle River, New Jersey, 1996.

[5] Richard C. Metzger, Brian VanVoorst, Luiz S. Pires, Rakesh Jha, Wing Au, Minesh Amin, David A. Castanon, and Vipin Kumar. *The C3I Parallel Benchmark Suite - Introduction and Preliminary Results.* Supercomputing '96, Pittsburgh, Pennsylvania, November 17-22 1996.

[6] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrel. *Pthreads Programming.* O'Reilly, Sebastopol, California, 1996.

[7] David J. Kruglinski. *Inside Visual C++.* Microsoft Press, Redmond, Washington, fourth edition, 1997.

[8] Thuan Q. Pham and Pankaj K. Garg. *Multithreaded Programming with Windows NT.* Prentice Hall, Upper Saddle River, New Jersey, 1996.

A Transparent Checkpoint Facility On NT

Johny Srouji
Johny.Srouji@intel.com
Paul Schuster
Paul.Schuster@intel.com
Maury Bach
Maury.Bach@intel.com
Yulik Kuzmin
Yulik.Kuzmin@intel.com

Intel Corporation, Israel Design Center

Abstract

With the increased use of networks of NT workstations for long-running engineering applications, process checkpointing and process migration can avoid wasted computer cycles and improve system utilization. The problem we solve is how to capture and reconstruct process state transparently and efficiently without affecting the correctness of the application.

A checkpoint facility enables the intermediate state of a process to be saved to a file. Users can later resume execution of the process from the checkpoint file. This prevents the loss of data generated by long-running processes due to program or system failures, and it also facilitates debugging when the bug appears after the program has executed for a long time.

This paper describes the implementation of a checkpoint library that permits users to save temporary state of long-running multi-threaded programs on a Windows/NT system and to resume execution from the checkpointed state at a later time. Our Windows implementation is the first such implementations that we are aware of for this operating system. Our implementation is portable, maintains good performance, and is transparent.

The checkpoint facility is currently used in several major internal projects at Intel.

1. Introduction

This paper describes a checkpoint facility for long-running programs on Windows/NT. The checkpoint facility permits users to save the state of a running process at arbitrary points of execution and to resume execution from the saved state at a later time. This facility is important for long-running processes, so that users can capture intermediate results of processes that do not run to completion because of machine or application failure. If a long-running processes is interrupted before completion but after a checkpoint, the user can resume execution from an intermediate state instead of having to re-run the process from start. Checkpointing also gives developers great leverage when debugging long-running processes; they can debug from a point deep in the run or change input data after run time, rather than having to restart programs from the beginning. Finally, checkpointing is an important milestone in the development of a facility for process migration, whereby processes can be halted on one machine, moved to another, and continue execution transparently.

Our checkpoint facility is non-intrusive to user programs, in that the programmer need not change any code to save checkpoints during execution. Users can resume execution from the point of the checkpoint and receive the same results they would have received without checkpointing, subject to changes in the run-time environment that may have occurred before the program is resumed. Of course, processes can continue to execute after completing a checkpoint.

Process migration improves the overall utilization of resources to achieve high performance, and enhance fault tolerance by providing the capability to move work from a failing machine.

The checkpoint system currently works on Windows/NT and on UNIX (AIX and FreeBSD) systems for several long-running simulation applications, but nothing precludes use of the checkpoint facility in other environments. This paper describes our NT implementation.

2. Design Goals

To ensure application transparency of process checkpointing, it is necessary to capture the process state and restore it later. Thus, the problem we solve is how to capture and reconstruct the process state efficiently without affecting the correctness of the application. Our implementation is portable, transparent to the user, and provides good performance.

The following high level design goals for the process checkpointing facility were followed:

- **Transparency**

  Our implementation does not require availability of user source to run the checkpoint utility. User applications need only link to the checkpoint library DLL, which will automatically change the startup routine and system call import table. Changing the startup routine allows us to inject optional checkpoint specific command line flags in the application and initialize checkpointing. Changing the system call import table allows us to wrap system API calls to preserve state across a checkpoint.

- **Correctness**

  Process execution gives the same results whether or not checkpoints are taken at runtime. Resuming a process from a checkpoint provides the same result as the original execution. We used a regression test suite that contains real application linkage as well as specific test cases to ensure that our checkpoint library is correct.

- **Minimal Performance Impact**

  The checkpoint facility writes the various memory segments of the application to a checkpoint file. Elapsed time is therefore directly proportional to process size. Figure 1 shows benchmark for a typical application sizes:

Process Size	Checkpoint File	Time to Checkpoint
10 MB	10648 KB	4 sec
20 MB	20888 KB	9 sec
30 MB	31128 KB	15 sec
50 MB	51608 KB	21 sec

Figure 1 - Checkpoint Performance

In our implementation, we wrap certain system and library API calls so that we can save state information. This adds a minimal overhead to the application. Figure 2 shows the average overhead for typical wrapped calls for 1 million consecutive calls of each function:

Library Call	Overhead
`CreateFile,CloseHandle`	6.8 x 10^{-5} sec
`WriteFile`	1.5 x 10^{-5} sec
`malloc, free` sequence	0 sec

Figure 2 - Checkpoint Overhead

- **Portability**

  Our checkpoint facility runs on Windows/NT, AIX and FreeBSD UNIX systems. We use a similar user-level methodology on all OS implementations, which has proved to be easily portable, but of course there are some code differences over OS implementations. Our NT implementation contains a few, small assembly routines written on Intel architecture, which are easy to port to other NT platforms. No application modifications are required.

- **Multiple Thread Support**

  Our solution supports checkpointing of multi-threaded Windows applications.

3. Architecture

The following block diagram provides a high level architecture of the checkpoint library.

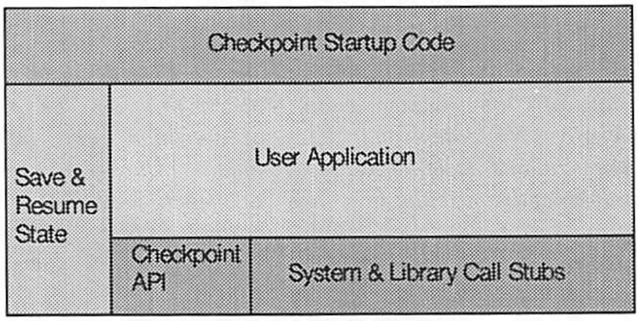


Figure 3 - Architecture

The library is implemented with two components:

1. Loader - a program that loads the user program into the operating system. We use this program because NT's virtual addresses allocations are affected by the length of the command line arguments. We need to ensure that the same virtual addressing is used at process resume regardless of the command line.

2. Checkpoint DLL - the main program that sets up wrappers to system and API calls in the user's inary, dumps the process state to a checkpoint file, and resumes execution of a process from a checkpoint file.

System and library API calls made by the application are redirected to versions provided by the checkpoint library. This allows system state held by the operating system on behalf of the process (such as open file handles) to be saved and recreated over a checkpoint. We use this "wrapper" method of saving library and system call state information, as it is portable between different operating systems.

When the operating system completes initialization of a user application, it loads the checkpoint DLL that changes the entry point of the application, so that checkpoint initialization routines will be called instead of the user program. The checkpoint DLL checks whether a process dump or resume is required. For a process dump, the checkpoint DLL updates the process import table to inject the API wrappers. Then the DLL creates a new thread for the program, which will be responsible for dumping the process and for timer notifications for periodic checkpointing. When all initialization procedures are completed, control is passed to the original user code entry point. For a process resume, the checkpoint DLL restores the state of the application when it was dumped, including threads, memory and system objects. If further checkpointing is required, the DLL proceeds as above; otherwise control is simply given to the resumed user code. Figure 4 shows a high level outline of the implementation. The checkpoint-specific arguments provide user run-time control over the checkpointing.

```
int
startup_algorithm ( )
{
   process_chkpt_args();


   if (checkpointing) {
      if (chkpt_periodic)
           set_chkpt_periodic(period);
       if (chkpt_on_signal)
           set_chkpt_on_signal(signum);
   }
   else if (resume)
      _chkpt_resume(dumpfile);

   return((int)
         _chkpt_user_main(argc,argv,envp));
}
```

Figure 4 - Startup Code

Checkpoint API calls are also available to the application programmer so that checkpoints can be requested at critical points in the application code.

As described above, linking an application to the checkpoint DLL library involves changing the application entry point. There is no need to recompile the user application or make any changes to the code.

4. Interface

The checkpoint facility provides a user-level interface and an API to the checkpoint capabilities.

4.1 Users Interface

The build process replaces the application's startup function with the checkpoint DLL startup code. As we have described, this startup code adds checkpoint specific options to the application so the user can control checkpoint behavior at run time. Regular application options are added after the checkpoint options.

The following checkpoint command line options are supported:.

- `loader <progname> -chkpt_P <period>`

checkpoint the process every *period* seconds

- `loader <progname> -chkpt_R <file>`

resumes process execution from the state previously checkpointed in *file*

- `loader <progname> -chkpt_D <dir>`

write checkpoint files to directory *dir*

- `loader <progname> -chkpt_X <file>`

extract header information from checkpoint *file*

4.2 Developer Interface

The checkpoint facility defines the following API call:

```
_chkpt_now (char *filename)
```

This function checkpoints process state to a file using the given *filename.* If passed a NULL string a standard sequential naming convention is used. In case of error, an external variable _chkpt_errno is set to the corresponding error number and may be used by the calling function.

5. Process State

Checkpointing and resuming a process should be transparent so that, as far as the application code is concerned, the checkpoint never happened.

To achieve transparency, the entire process state must be captured at checkpoint time and fully restored when the process is resumed. Figure 6 shows the components of typical process state, which we explain from the bottom up.

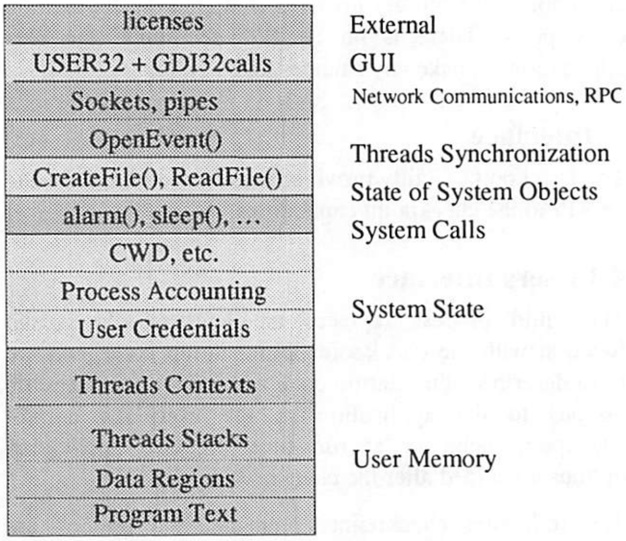


Figure 6 - Process State

Program text is the application object code, which is typically memory mapped from the executable file on disk. Data regions include statically and dynamically allocated data, for example calls to `malloc` or `new`. Program stack and the value of registers, stack pointer and program counter complete the memory components of the process state.

System state is held by the operating system on behalf of the process. It includes user credentials such as the process ID and process owner, various accounting data (for example cumulative CPU usage), and other miscellaneous state such as the current working directory.

At runtime, the process may use system calls that maintain state, for example `Sleep` suspends a thread for a specified time interval. If in the meantime the process checkpoints, the checkpoint DLL must capture this state. An interesting problem with `Sleep` is whether resume should honor the original request with respect to real time or process run time at resume. Our implementation assumes process run time, although the true definition of this API call is real time.

A process may have open files and inter-process communication channels with data in transit at checkpoint time. Graphics applications will have additional state; for example a graphic application may have called the Win32 API `GetWindowDC()` to obtain a device context. Finally some processes such as those that may have requested a floating license will have state held in some external entity.

As we move up the layers of Figure 6, process state becomes progressively harder to capture. A basic checkpoint implementation can be built by capturing only the lower user memory layers. However, a practical checkpoint facility must capture at least parts of the system state, system calls and open files.

6. Implementation

We now describe the implementation of the process checkpoint facility.

6.1 Checkpoint Initialization

The checkpoint library first needs to assume control of the program at execution time to check for any specific command line arguments and to create a special checkpoint control thread.

We use a self-modifying code technique to change the executable's entry point to gain control transparently at program startup. When a program is executed in NT, the system library calls each DLL's initialization routine (**DllMain**) before passing control to the program's entry point. The checkpoint DLL's initialization routine locates the program entry point, saves and then overwrites the existing instructions with a `jmp` instruction to a checkpoint specific startup function. The entry point is easily extracted from the in-memory process image header which can be located using the `GetModuleHandle()` Win32 API.

The checkpoint startup function runs as the program entry point and parses the command line using the global `__argc`, `__argv` symbols. If checkpoint or resume mode is requested a special checkpoint control thread is created. When running in checkpoint mode, the saved program entry point instructions are restored, checkpoint command flags are stripped and another `jmp` instruction is executed back to the original program entry point. Control passes to the user code.

6.2 Checkpoint Control Thread

The special checkpoint thread is responsible for controlling the process checkpoint (dump) and resume. This thread continuously scans its APC (Asynchronous Procedure Call) queue waiting for dump or resume requests to arrive. APC queue scanning is implemented in the OS core and does not take additional application cycles.

The library supports two types of checkpointing: periodic and on-demand (through a checkpoint library API call). In periodic mode a waitable object is created and each time the wait period elapses, the dump APC is placed on the checkpoint thread's APC queue. With on-demand mode, the checkpoint API `_chkpt_now()` function call creates a dump APC with 0 wait period.

6.3 Process Dump

On receipt of a dump APC call, the checkpoint thread suspends all other program threads and saves their context by calling `GetThreadContext()`, then opens a new checkpoint file. A naming sequence is used to ensure that filenames are unique, and that the checkpoint order can be determined even when a checkpoint is taken in a process that was itself resumed from a checkpoint.

A header is written to the file with global information about the process such as the process name, current working directory, time of checkpoint and the command line arguments. Process state is saved to the file as described below. The checkpoint thread then resumes all suspended threads simultaneously before putting itself into a blocking wait on the next APC queue event.

The following sections describe how we capture state of the individual components of the process.

- **Text Segment**

We assume the original object file is available at resume, and so the program text segment does not need to be saved. We check time stamps on process resume to verify that the original object file has not changed.

- **Data Segments**

The checkpoint routines write the contents of the process data segments to the checkpoint file. The difficulty is to identify the start and end virtual addresses for each segment.

Static data regions are allocated when the process image is loaded. The base address of the process image is located using the `GetModuleHandle()` Win32 API. The static data regions can be located from this base address with sequential calls to the `VirtualQuery()` Win32 API. All writeable regions that have the same base allocation address contain static data. We prefer this technique over the simpler method of extracting location information on static data regions from the image header, because it is more reliable: A developer can add and change static region names at link time, making them impossible to identify.

We make an extra check, so that we do not include the static import table data, as this has been modified by our checkpoint DLL (see Section 6.4.2) The import table address and size can be extracted from the in-memory process image header.

Dynamic data segments are allocated in several ways. Heap space is allocated with `HeapAlloc()`, `GlobalAlloc() and LocalAlloc()` Win32 API functions. Heap allocations can be located using the `GetProcessHeaps()` Win32 `API`. For efficiency and to avoid problems at resume, all heaps expect for that used by the checkpoint DLL (allocated by the CRT library) are saved. The checkpoint DLL heap can be identified using the address of the global symbol `_crtheap`.

Dynamic memory allocations through calls to the `VirtualAlloc()` Win32 API are redirected to a checkpoint wrapper stub, which saves all address allocation information.

- **Thread Execution Context and Stack Segment**

The execution context of each thread was saved at the beginning of the dump sequence and is simply written to the checkpoint file along with the contents of each thread's stack.

The end address of each stack is located using the stack pointer (ESP) contained in the thread execution context. Appropriate calls to `VirtualQuery()` are used to locate the start address and size of the stack.

- **System State**

Information needed to reconstruct system state changed by Win32 API calls is written to the checkpoint file. This information is captured through our technique of redirecting Win32 API calls through wrapper functions discussed in Section 6.5. For each wrapped API we save the call parameters, thread ID and any call specific information.

6.4 Process Resume

An application resumes by reconstructing its state, using a previously created checkpoint file. At startup the checkpoint DLL checks for the resume command line parameter. If present, a resume APC is placed on the checkpoint thread's APC queue.

The checkpoint thread opens the file that contains the process checkpoint data and examines the file header to ensure the checkpoint was created by a prior invocation of the currently running object code. It then reads data from the checkpoint file and reinstates system state including thread creation, execution context, stack and data segments. After state has been restored, all threads are simultaneously resumed and the process continues from where it was checkpointed. We discuss these steps in more detail below.

- **Process and Thread Creation**

Process creation is implemented by invoking the original application with the `-chkpt_R` argument, which restores the original text segment and transfers control to the resume APC in the checkpoint thread.

The checkpoint thread reads saved Win32 API call state information from the file and uses the information on calls to the `CreateThread()` Win32 API routine to create a new set of threads. New and old thread ID's are saved in a special association table, allowing for translation by subsequent calls through Win32 API function wrappers.

Each newly created thread is put in a mode similar to the checkpoint thread, continuously waiting for APC's to execute.

- **System State**

For each saved Win32 API call, we use the thread association table to map the original calling threadID to a new thread. The original parameters and name of the API function to call are sent to the thread using `QueueUserAPC()`. The thread executes the appropriate Win32 API function. API calls are sent to the threads in the same order they were originally called.

After all saved Win32 API calls have been re-invoked, a special APC is sent to each thread, which respond by executing `WaitForSingleObject()` on a common event object. This object is used to wake all threads simultaneously once all process state has been recovered.

- **Data Segment and Thread Context Resume**

The checkpoint thread now restores data and stack segments. Data is read from the checkpoint file and written directly to memory to the saved addresses of each process region. When all memory related data is recovered, thread contexts are restored using the `SetThreadContext()` API.

- **Process Control Resume**

At this point all state has been restored. All threads are waiting on a single event object and their contexts, including their instruction pointer, are set to the checkpointed values. The checkpoint thread sets the common event object and all the threads simultaneously resume from where they were suspended by the original dump APC. Finally, the checkpoint thread puts itself to sleep, and waits for further Dump APC calls.

6.5 System Calls

Some Win32 API calls such as `CreateThread()`, `CreateFile()`, and `CreateSemaphore()` change system state for a process. These API calls present a problem for checkpointing, since they represent state that is held within the operating system on behalf of a process. Without privileged access to kernel data space, this state is difficult to capture and restore.

To remove the need for privileged kernel access, we adopt a technique of redirecting certain Win32 API calls through function wrappers. The wrappers save enough call information that we can recreate state upon resume. For example knowing the name of an open file and the current offset, we can reopen the file during resume, seek to the saved offset, and craft the original file handle to correctly access the reopened file in all subsequent Win32 API calls.

It is important to do this at the Win32 API interface (KERNEL32.DLL) boundary so that we do not need to deal individually with the thousands of library functions provided by the NT programming environment. For example by wrapping the `VirtualAlloc()` Win32 API call we do not need to deal with the `malloc` family of library functions.

6.5.1 Redirecting Win32 API Calls

In Windows NT, each API call is redirected by the linker to the IAT (Import Address Table), from where another jump is taken to reach the real API function handler. The basic technique of wrapping Win32 API calls is to change the addresses in the IAT, so the second jump will lead to the appropriate checkpoint wrapper, which will collect necessary information about the call before the real API is executed.

One method to do this would be to simply redirect the call to our routine by changing the symbol. Our routine could then call the Win32 API directly. However this method does not work because the checkpoint routine adds a frame to the stack, and the real API will get an incorrect stack pointer. Registers may also be changed by the checkpoint wrapper routine. To avoid this problem, we again used a self-modifying code technique. For each wrapped Win32 API, a small code fragment is created at run-time, where we save the registers and execute the corresponding checkpoint wrapper function. After the wrapper returns, the registers are restored, and control is passed to the real API handler through a `jmp`.

Since the real Win32 API function is now called from the same stack frame as if it was called directly, the checkpoint wrapper redirect is transparent to the user code and the Win32 API function.

6.5.2 Supported System Calls

We support the following important Win32 API functions in the current version of the checkpoint library:

EnterCriticalSection, InitializeCriticalSection, LeaveCriticalSection

CreateThread, GetStdHandle

HeapCreate, HeapAlloc, VirtualAlloc

CreateFile, ReadFile, WriteFile, CloseHandle

Figure 7 – Checkpointed Win32 API Functions

Another 26 functions are tested, and could be used with the checkpoint facility.

7. Comparison to Similar Work

There are several existing solutions to the checkpoint and process migration problems on UNIX, and we choose to discuss a few of them. We could not find any similar work on NT.

7.1 MPVM

MPVM, an extension of *PVM,* is a research project at Oak Ridge National Laboratory that allows parts of a parallel computation to be suspended and subsequently resumed on other workstations by migrating process state from one machine to another. Migration transparency is addressed by modifying PVM libraries and daemons and by providing wrapper functions to certain system calls so that the migration occurs without modifying the application code. The migration mechanism is implemented at user level. *MPVM* has the following limitations:

- The developer must explicitly create executable files that are statically linked to support shared libraries.
- *MPVM* assumes use of a global file system.

7.2 Condor

Condor is a batch facility running on UNIX systems that allocates processes to idle work stations. It performs process migration by checkpointing a process to a file, transferring the file, and then restoring the process from the checkpoint file. No special programming is required, but user applications need to be re-linked with *Condor* libraries.

Condor has several limitations:

- Condor does not support all system calls and library calls. Signals and signal handlers are not supported, and *popen* is not supported..
- All file operations must be idempotent - read only and write only file accesses work correctly, but programs which read and write a same file may not checkpoint transparently.

7.3 Libchkpt

Libckpt shares many goals of our checkpoint facility, but they do not support a complete set of system wrappers. They support file system calls such as open, close, read and write, but they do not support popen, or signals. Our UNIX version of the checkpoint facility does support popen and signals.

7.4 MOSIX

MOSIX is a multi-computer operating system that supports transparent, preemptive process migration and load balancing for efficient utilization of overall resources and to balance work distribution. The MOSIX enhancements are implemented at the operating system kernel level without changing the UNIX interface, and therefore it is completely transparent to the application level. It uses *PVM* as the distribution engine. The process migration in MOSIX is dynamic and preemptive, that is it responds to variations in workstation load by migrating processes from one node to another at any stage of the life cycle of a process. The granularity of the work distribution in MOSIX is the UNIX process. The processors must be homogeneous (from the same family) to allow process migration. MOSIX has the following limitations:

- It is implemented at the kernel level and therefore it is more complex and requires source code availability.
- It is a preemptive system that does not provide the capability to dump/resume user processes at arbitrary times.

8. User Experience

Many internal Intel simulators and program environments use the checkpoint facility on Windows NT and UNIX (AIX and FreeBSD) systems. Developers typically checkpoint their work every hour, permitting them to focus on a problem area that is discovered deep into a run. When a bug is discovered, they resume execution from the nearest checkpoint, sometimes set up finer-grained checkpoints, and have been able to debug their programs with greater ease than was the case before use of the checkpoint facility.

9. Limitations

The following limitations exist in our current checkpoint implementation.

9.1 External File Persistence

Where the runtime environment of a program depends on some external data such as a file or network connection, the checkpoint facility cannot correctly restore state if the corresponding media is unavailable or modified at process resume. For example if some input file is deleted after a checkpoint is taken, the resume process will be unable to reopen the file and recreate the file handle that may be necessary at a later stage in the process execution.

9.2 Direct System Object Access

The current design of the checkpoint utility cannot save the state of system objects that were changed in a way other than calling Win32 system APIs (provided by KERNEL32.DLL). For example:

- The checkpoint utility can't set API wrappers on dynamically loaded system APIs that are called using the pointer returned by `GetProcAddress()`. Those calls don't go through the import table, and so our current method of wrapping calls will not work.
- Checkpoints may fail if a user program calls undocumented NTDLL.DLL services or calls "`int 2Eh`" to get system services.
- We cannot checkpoint kernel mode drivers or programs that dynamically modify kernel drivers.

9.3 Direct System Data Access

The checkpoint facility assumes that user programs do not manually modify system data that is maintained in virtual space of the program by system DLLs and the operating system. For example: the data segment of KERNEL.32 DLL. Such modified data will not be restored.

10. Future Development Plans

10.1 Optimization

The current version of the checkpoint system dumps the entire process state for each checkpoint call. This requires time proportional to the size of the process, and can clearly be an expensive operation for large processes. The developers of the Libchkpt checkpoint system [4] note that it is possible to speed up checkpointing procedures by dumping only those pages whose data has changed since the previous checkpoint call. We may optimize our checkpoint facility to dump incremental changes to the process state by using the `VirtualProtect` API to write-protecting pages using the GUARD_PAGE facility to mark the page as accessed, then writing only marked pages at the next checkpoint call.

10.2 Multithreaded API Call Support

As discussed in section 6.5, we support checkpoint of applications that use system API calls by call redirection through our code, which captures calling thread ID, call parameters and return values. We then use this data to reconstruct system state by re-calling the system API functions from the context of the original thread at process resume. At checkpoint, all threads are suspended regardless of whether or not they are in the middle of a system API call.

This technique can be problematic as demonstrated in the following example. Suppose a program thread calls the `HeapAlloc()`API to allocate heap memory. This API will reserve a memory segment for the user data and write some meta-data structures to memory for managing the allocation. If we suspend the thread in the middle of this API, the checkpoint thread will have recorded that there was a call to `HeapAlloc()` and the checkpoint routine that saves memory segments may write the allocated segments to the checkpoint file before the `HeapAlloc()` API has finished writing the meta-data.

Upon resume, we will restart all the API calls, and `HeapAlloc()` will be called and setup the correct memory segment and meta-data structures. However the checkpoint code will then restore memory segments, which will overwrite the good meta-data structures with the in-complete saved structures.

We need to get a better solution to restore threads that were check-pointed part way through system API calls. One possible direction is to write an NT kernel driver which would be able to save the exact instruction at the time of checkpoint and at resume place a breakpoint at the same instruction. When all state has been restored and the program threads are resumed, they would continue at the exact point where they were check-pointed.

10.3 Enhanced System Calls Support

The type of applications the checkpoint library supports are heavily dependent upon the range of system API calls that can be supported over a checkpoint resume operation. In our work we support system calls used by a number of internal applications.

Of particular interest, but highly difficult to implement, is support for system API calls that involve multiple processes such as those used for named pipes, Windows sockets, RPC, and COM Interfaces. It is difficult to checkpoint several processes simultaneously and to capture data that may be in transit between processes (such as on a network), and later restore state transparently. Some distributed process checkpoint research has been done, but is generally application specific.

11. Conclusion

Intel engineers run many simulators for research, specification and validation of new chips. Simulators typically run a long time, sometimes for weeks, to produce results. It is not uncommon for simulations to fail to complete because of system crashes, environmental failures such as electricity outages or, last but not least, programmer bugs. Employment of checkpoint procedures minimizes the costs of these failures by retaining results of a run until the last good state before failure, by permitting execution to restart from an advanced point of execution rather than from the beginning of the run, and by providing an advanced state for debugging that permits programmers to correct their programs more easily for future long-run use.

This paper described the implementation of a checkpoint facility that is being used in applications that run on NT systems. The checkpoint facility is a general purpose library that can be linked and used with many applications, saving developers the need to develop ad hoc solutions to checkpoint their programs. The checkpoint facility runs transparently to the application, and programmers do not have to change source code to obtain process checkpoints. Users have great flexibility in choosing the names and locations of checkpoint files and the frequency and circumstances under which checkpoints are created.

Our users have reported great success with our checkpoint facility, primarily in debugging long-running processes.

Acknowledgment

We would like to thank those who supported and used our development of checkpoint facility: Avi Giora, Shalom Goldenberg, Ariel Berkovits, Yosi Mor, Amit Dagan, Yaron Sheffer and Eric Koldinger.

References

[1] Jeremy Casas, et al. MPVM - A Migration Transparent Version of PVM. Computing Systems, vol. 8, no. 2, pp. 171-216, Spring 1995.

[2] Allan Bricker, Michael Litzkow, and Miron Livny: Condor Technical Summary, Version 4.1b, *University of Wisconsin - Madison, 1991.*

[3] Barak A., Braverman A., Gilderman I. and La'adan O., Performance of PVM with the MOSIX Preemptive Process Migration, *Proc. 7th Israeli Conf. on Computer Systems and Software Engineering, Herzliya, pp. 38-45, June 1996.*

[4] J.S. Plank, M. Beck, and G. Kingsley, Libckpt: Transparent Checkpointing Under Unix, *1995 Usenix Conferencee.*

[5] M.J. Litzkow. Remote UNIX: Turning Idle Workstations into Cycle Servers. In *Proc. USENIX summer '87, Phoenix, Arizona, June 1987.*

[6] K.I. Mandelberg and E. Sunderam. Process Migration in UNIX Networks. In *Proc. USENIX Winter '88, Dallas, Texas, February 1988.*

[7] R. Alonso and K. Kyrimis. A Process Migration Implementation for a UNIX System. In *Proc. USENIX Winter '88, Dallas, Texas, February 1988.*

[8] M.J. Litzkow, M. Livny, and M.W. Mutka. Condor - A Hunter of Idle Workstations. In Proc. 8th Int. Conf. On Distributed Computing Systems, San Jose, California, June 1988.

[9] James S. Plank, Micah Beck, and Gerry Kingsley. Libckpt: Transparent Checkpointing under UNIX. In *Proc. USENIX Winter 1995, New Orleans, Louisiana, January 1995.*

Merging NT and UNIX Filesystem Permissions

Dave Hitz (hitz@netapp.com)
Bridget Allison (bridget@netapp.com)
Andrea Borr (aborr@netapp.com)
Rob Hawley (hawleyr@netapp.com)
Mark Muhlestein (mmm@netapp.com)

Network Appliance (www.netapp.com)

Abstract

Sharing network data between NT and UNIX systems is becoming increasingly important as NT moves into areas previously serviced entirely by UNIX. One difficulty in sharing data is that the two filesystem security models are quite different. NT file servers use access control lists (ACLs) that allow permissions to be specified for an arbitrary number of users and groups, while UNIX NFS servers use traditional UNIX permissions that provide control only for owner, group, and other. This paper describes an integrated security model in which a single filesystem can contain both files with NT-style ACLs and files with UNIX-style permissions. For native file service requests (NT requests to NT-style files and NFS requests to UNIX-style files) the security model exactly matches an NT or UNIX file-server. For non-native requests, heuristics allow a reasonable level of access without compromising the security guarantees of the native model.

1 Introduction

Network Appliance file servers support the native file service protocols for both NT and UNIX (CIFS for NT and NFS for UNIX), but until now, the filer's underlying security model has been based on UNIX. This paper describes a new version of NetApp's system software that supports UNIX permissions as well as NT access control lists (ACLs).

See [Hitz95], [Watson96] and [Hitz94] for details on the NetApp filer and the WAFL filesystem that it uses. For the purposes of this paper, a brief summary will suffice.

NetApp filers are dedicated devices, or "appliances," that perform just one function. Just as a Cisco router is a dedicated device optimized entirely for routing, a NetApp filer is optimized entirely for file service. We believe that an appliance that performs just one function can be faster, simpler, and more reliable than a general-purpose computer performing that same function.

NetApp's WAFL filesystem is designed to accommodate multiple filesystem protocols. When accessed via NT, for instance, WAFL does case-insensitive name lookups, but for NFS it does case-sensitive lookups. The on-disk inode structure for each file includes both UNIX metadata, such as the UNIX-permissions, as well as NT metadata, such as the hidden and archive bits. Similarly, the on-disk directory format includes both a long file name and a DOS-style eight-dot-three file name.

Filers can be administered by either NT or UNIX system administrators. NT administrators can use familiar tools such as Server Manager and User Manager. For UNIX administrators, there is a command line interface with UNIX-like commands such as **ifconfig**, **nfsstat**, and **ping**. There is also a browser-based GUI, written in Java.

2 Design Overview

The new integrated security system was designed to meet three goals:

(1) Make NT/Win95 Users Happy.

Support an NT-centric security model based on ACLs. To an NT or Win95 client, this security model should behave exactly like an NTFS filesystem on an NT file server.

(2) Make UNIX Users Happy.

Support a UNIX-centric security model based on UNIX permissions. To an NFS client, this security model should behave exactly like a UNIX NFS server. (This is the only security model that NetApp filers have historically supported.)

(3) Let NT and UNIX Users Share Data.

Provide reasonable heuristics so that NT/Win95 users can access UNIX-style files, and UNIX users can access NT-style files.

To meet these goals, NetApp allows administrators to designate specific sections of the filesystem as an NTFS-qtree or a UNIX-qtree. (A qtree is simply a designated subtree within the filesystem.) NT and UNIX users can safely access both types, but the NTFS-qtree seems more natural for NT users, and the UNIX-qtree for UNIX users.

Native requests – NT networking to an NTFS-qtree or NFS to a UNIX-qtree – work exactly as expected. UNIX-qtrees are modeled after Solaris, and NTFS-qtrees are modeled after NT. Non-native requests use heuristics designed to operate as intuitively as possible while still maintaining security.

Non-native NT requests are handled by mapping the NT user to an equivalent UNIX user, and then validating against the standard UNIX permissions. In the simplest case, John Smith might have the account "john" on both NT and UNIX. When John accesses a UNIX-qtree from NT, the filer looks up "john" in /etc/passwd (or over NIS), and uses the specified UID and GID for all access validation. A user-mapping file handles the case where John's NT account is "john", but his UNIX account is "jsmith". NT users with no UNIX account may be mapped to "nobody", to a specified UNIX account, or they may simply be denied access.

NFS requests to NTFS-qtrees are validated using special UNIX permissions that are set whenever an ACL is updated. The UNIX permissions are guaranteed to be at least as restrictive as the ACL, which means that users can never circumvent ACL-based security by coming in through NFS. On the other hand, since UNIX permissions are less rich than NT ACLs, a multiprotocol user may be unable to access some files over NFS even though they are accessible via NT. In practice, NFS access to NTFS-qtrees works well for the owner, and for access granted to the NT "everyone" account, but not for other cases. (These restrictions will be removed in a future release, as described in section 7.)

The filer also supports a Mixed-qtree in which the security style is determined on a file-by-file basis. Files created by NT users get NT ACLs, and files created by UNIX users get UNIX permissions. A file's security style may be changed from one style to another by NFS "set attribute" or NT "set ACL" requests, assuming – of course – that the requestor has the appropriate permissions. This style is ideal for users who actively use both NT and UNIX and want access to both styles of security.

Several features allow special control over privileged users. The filer supports the NT "administrators" local group, which lists the NT accounts that have administrator privileges. The NT User Manager interface can be used to manage the "administrators" local group over the network. The user-mapping file can be used to map the NT "administrator" account into UNIX "root", or to a non-privileged UNIX account such as "nobody". Privileges for UNIX "root" are controlled using the /etc/exports "root=" flag. Requests from "root" are mapped to "nobody" unless the "root=" flag on an export explicitly allows root privileges.

3 Background

This section briefly describes UNIX and NT filesystem security, since many people are familiar with one or the other, but not both.

3.1 UNIX Filesystem Security

UNIX uses user IDs (UIDs) to identify users, and group IDs (GIDs) to identify groups. The permission bits themselves control read access (r), write access (w), and execute access (x). The full set of UNIX permission information stored with each file consists of:

- UID of the owner
- GID of the owner
- User perm bits (controls rwx for owner)
- Group perm bits (controls rwx for the group)
- Other perm bits (controls rwx for anyone else)

When performing validation, UNIX determines whether the request is from the file's owner, someone in the file's group, or anyone else, and then uses the appropriate permission bits.

See [McKusick84] or [Bach86] for more details.

3.2 NT Filesystem Security

NT uses security IDs (SIDs) to identify both users and groups. The NT permissions for each file consist of:

- SID of the owner
- SID of the owner's group
- ACL (Access Control List) for the file

The ACL contains one or more access control entries (ACEs). Each ACE contains a SID, indicating the user or group to which the ACE applies, and a set of permission bits. NT permission bits include the three UNIX bits – read, write, and execute – as well as "change permissions" (P), "take ownership" (O), "delete" (D), and others. An ACE can either grant or deny the specified permissions. One ACE might grant read and write permission to the engineering group, but another ACE might specifically deny write permission to John Smith. Even if John is in the engineering group, he will be denied write access.

NFS and NT also differ in how they authenticate users. NFS is a connectionless protocol, and each NFS request includes the UID and GIDs of the user making the request. The UNIX client determines the UIDs and GIDs when the user first logs in, by looking at the files /etc/passwd and /etc/groups. NT networking is session based, so the identity of the user can be determined just once, when the session is first set up. At session connect time, the client sends the user's login name and encrypted password (actually the challenge and the client's response) to the file server, and the server determines the session's user SID and group SIDs. Servers commonly forward the name and password to an NT domain controller (DC) and let the DC perform authentication.

See [Reichel93] for more details.

4 Philosophy

4.1 Surprise and Insecurity

Given that NT and UNIX have fundamentally different models of filesystem security, it is impossible to design an integrated model that performs exactly as expected for all users. NT ACLs provide a richer security model than UNIX permissions. Many NT ACLs cannot be accurately reflected to a UNIX user. Yet, when a UNIX user lists a directory, NFS must return something for the UNIX permissions.

We conclude that any integrated filesystem security model must present users with some combination of surprise and/or insecurity. If we validate UNIX requests directly against the NT ACL, then the UNIX user may be surprised to see behavior that's different than what the faked-up UNIX permissions would seem to imply. But if we validate UNIX requests against faked-up UNIX permissions, then this may result in insecurity if the UNIX permissions grant more access than the NT ACL, or surprise if the UNIX permissions deny access where the NT ACL would have granted it.

When a filesystem accepts a request to set security (NT or UNIX) on a file, it has – in essence – made a promise to the user. Violating this promise is little different than losing data that a user thought was safely written. Hence, NetApp's implementation allows no insecurity, and it minimizes surprise as much as possible given this constraint.

Some users may dislike surprise more than they dislike insecurity. In a mixed NT and UNIX development environment, making the development tools work is probably the most important goal. Sacrificing security may be acceptable. (At NetApp, the software developers all have privileged access anyway.) Perhaps someday we will add options to control the balance between surprise and insecurity, but for the first release, erring towards safety seemed best.

In examining NetApp's integrated security model, it is important to remember that no perfect solution is possible. We must be willing to make trade-offs between various types of surprise, and – for sites willing to allow it – perhaps even between surprise and insecurity.

4.2 Which to Map: Users or Permissions?

As described in the Design Overview, NetApp handles non-native NT requests by mapping the NT user into an equivalent UNIX user, and validating requests directly against the UNIX permissions. We call this *user mapping*.

On the other hand, NetApp handles non-native NFS requests by using faked-up UNIX permissions that are set whenever an NT ACL is updated. We call this *permission mapping*.

We believe that user mapping reflects the intended security more accurately than permission mapping. The security models of NT and UNIX are so different that permission mapping can never be completely accurate. However, the two operating systems have very similar definitions of a user, so user mapping is straightforward.

We use permission mapping for non-native NFS requests simply because it is easier and cheaper. Permissions need be mapped only when an ACL is set or updated, which is rare. User mapping would need to be performed for every single NFS request. Unlike NFS, which is stateless, NT's CIFS protocol is session based, so user mapping need be done only once, when the session is established, rather than for each separate request.

Although we didn't implement it in our first ACL release, we now believe that with appropriate caching, it should be possible to do UNIX to NT user mapping efficiently. Section 7 describes our plans.

4.3 Issues for Non-Native Security

As described above, a native request is an NT request to an NTFS-qtree, or an NFS request to a UNIX-qtree. A non-native request is the reverse: NT to UNIX-qtree or NFS to NTFS-qtree.

This section examines the issues that are important for non-native requests, and it provides the outline for sections 5 and 6 below, which discuss how non-native NT and non-native NFS requests are handled.

The primary function of any filesystem security model is to validate requests – to accept them or deny them based on the authenticated user and the permissions for the file. Thus, validating non-native requests is one important topic, and is covered in 5.1 and 6.1.

In addition, there are several file system actions that require special attention. In particular, we must ask:

- How are requests to display permissions handled?
- How are requests to set permissions handled?
- How are permissions set for newly created files?

These are covered in 5.2 and 6.2.

5 Handling Non-Native NT Requests

This section describes how NetApp filers handle NT requests to UNIX-style files. Remember, files with UNIX permissions occur both in UNIX-qtrees, where all files are UNIX-style, and in Mixed-qtrees, which have both UNIX-style and NT-style files.

Section 5.1 discusses how non-native NFS requests are validated, and section 5.2 discusses non-native handling for displaying permissions, setting permissions, and creating new files.

5.1 Validation

NetApp filers validate NT requests to UNIX-style files by generating a mapped UID (and GIDs) for each NT networking session, and then using the UID (and GIDs) to check against the UNIX permissions.

Suppose that the NT user "john" connects to a filer. Here are the steps that the filer takes to determine the mapped UID and GIDs for "john".

(1) The filer sends a request to the NT domain controller (DC), to authenticate "john", and to find the NT SID for "john".

(2) The filer looks in the user-mapping file to determine whether the NT account "john" maps into a different account name under UNIX. In this case, let's assume that "john" maps into the UNIX account "jsmith".

(3) The filer looks up "jsmith" in /etc/passwd (possibly via NIS) to determine the UNIX UID and primary GID for John.

(4) The filer uses /etc/groups (possibly via NIS) to determine the UNIX GIDs for John.

These steps provide each NT networking session with a full set of UNIX authentication information, which allows the filer to easily validate most requests against the UNIX permissions.

Some NT operations don't map well to UNIX operations, so they must be handled specially:

- **Set ACL**

  The NT "set ACL" operation (similar to UNIX chmod) is always denied in UNIX-qtrees. In Mixed-qtrees, a "set ACL" operation is only allowed by the owner – that is the mapped UID for the NT session must match the file's UID. This operation converts the file from UNIX-style permissions to NT-style permissions.

  Only the owner can set an ACL because in UNIX only the owner is allowed to set attributes. (Remember, we are discussing requests to UNIX-style files.)

- **Take Ownership**

  The NT "take ownership" operation (similar to UNIX chown) is always denied in UNIX-qtrees. In Mixed-qtrees, only the file's owner can "take ownership" of a UNIX-style file. Like the "set ACL" request, this converts the file to NT-style permissions.

5.2 Request Processing

This section considers non-native NT requests in light of the three questions listed above, in Section 4.3, Issues for Non-Native Security:

- How are requests to display permissions handled?
- How are requests to set permissions handled?
- How are permissions set for newly created files?

5.2.1 Displaying Permissions

For non-native NT requests to display permissions, WAFL dynamically builds an ACL designed to represent the UNIX permission as well as possible.

On might hope to build an NT ACL that perfectly represents the UNIX permission like this:

- Owner – map the file's UID into an NT SID
- Group – map the file's GID into an NT SID
- ACE for owner SID – based on UNIX user perms
- ACE for group SID – based on UNIX group perms
- ACE for special NT *everyone* SID– based on UNIX other perms

Unfortunately, we currently have no way to map UIDs or GIDs into SIDs, so this approach isn't possible. Instead, we construct an ACL using only well known NT SIDs and the SID for the NT networking session itself. These are sufficient to let us construct an ACL that, while not perfect, does provide useful information.

Each faked-up ACL contains two ACEs (access control entries):

- ACE for NT "everyone" SID – based on the UNIX other permissions.
- ACE for the SID of the NT networking session – based on whichever UNIX permission is appropriate. If the mapped UID for the session is the file's owner, the ACE is based on the UNIX owner perms. If the group matches, then it's based on the group perms. Otherwise it's based on the other perms.

Note that this faked-up ACL always contains an entry for the user making the request, so users can always determine their own access rights.

If the NT session owns the file, then in the faked-up ACL the session's SID is shown as the owner. If not, then the well known NT SID "CREATOR_OWNER" is shown as the owner.

5.2.2 Setting Permissions

In UNIX-qtrees, NT requests to set permissions are always denied.

Outside of UNIX-qtrees, non-native requests to set permissions are allowed only by the file's owner. If allowed, the specified ACL takes effect just as it would have if the file had already been an NT-style file. After the "set ACL" request is processed, the file becomes an NT-style file.

5.2.3 Setting Permissions on Create

Unlike UNIX, which passes the permissions for a new file as part of the create request, NT expects permissions to be inherited from the parent directory.

The filer handles NT create requests in UNIX-qtrees as follows:

- The file's owner is set to the mapped UID for the NT networking session.
- The file's group is set to the mapped GID for the NT session, or inherited from the parent directory if the directory's SGID bit is set.
- The UNIX permission bits are inherited from the parent directory, except that SUID and SGID bits are cleared for non-directory creates.

In Mixed-qtrees, the newly created file inherits NT ACLs if the parent is an NT-style directory, but it inherits UNIX permissions, as described above, if the parent is a UNIX-style directory.

6 Handling Non-Native NFS Requests

This section describes how NetApp filers handle NFS requests to NT-style files.

Section 6.1 discusses how non-native NFS requests are validated, and section 6.2 discusses non-native handling for displaying permissions, setting permissions, and creating new files.

6.1 Validation

Whenever an NT ACL is set or changed, WAFL calculates a corresponding set of UNIX permissions. As a result, very little special processing is required to validate NFS requests to NT-style files. Simply doing the normal checks against the UNIX permissions usually does the right thing. The rest of this section describes how the UNIX permissions are constructed from the ACL, and explains a few special exceptions.

Converting an NT ACL into UNIX permissions is surprisingly tricky. This section gives a brief overview, but an observant user may encounter slight differences in the actual implementation.

- The file's UID is set to the mapped UID for the NT session.

  (Remember that the faked-up UNIX permissions are generated right when the ACL is set, so it makes sense to set the owner of a newly created file to the mapped UID for the NT session.)
- The file's GID is set to the mapped GID for the NT session.
- The UNIX user perm is set based on the access rights that the ACL grants to the NT session creating the file.
- The UNIX other perm is set based on the access rights granted to the NT "everyone" account. (If the ACL contains any denies, then the denied permissions are subtracted from the other perms.)
- The UNIX group perm is set equal to the other perm.

This design avoids security holes by ensuring that the UNIX permission is always at least as restrictive as the NT ACL. Unfortunately, UNIX permissions cannot represent the full richness of the NT security model. As a result, a file that a user can reach via NT may not be accessible via NFS.

Because NT supports some specific permissions that UNIX lacks, it is not possible to rely entirely on the UNIX permissions to validate some NFS requests:

- **REMOVE/RMDIR**

  Only the owner of a file is allowed to delete it. This is necessary to avoid violating the NT "delete child" permission.

- **CREATE**

  Only the owner of a directory can create anything in it. This is required in order for NT ACL inheritance to work properly. (This is explained more fully below, in section 6.2.3.)

Both of these restrictions will be removed by the future enhancements described in section 7.

6.2 Request Processing

This section considers non-native NFS requests in light of the three questions listed above, in Section 4.3, Issues for Non-Native Security:

- How are requests to display permissions handled?
- How are requests to set permissions handled?
- How are permissions set for newly created files?

6.2.1 Displaying Permissions

As described above, in section 6.1, every file with an NT ACL also has a set of UNIX permissions stored with it. To handle an NFS "get attributes" request, the filer simply returns those stored permissions.

6.2.2 Setting Permissions

In NTFS-qtrees, NFS requests to set permissions are always denied.

In Mixed-qtrees, only a file's owner is allowed to set permissions. When UNIX permissions are set on a file, the NT ACL is deleted – the file changes from NT-style to UNIX-style.

Note that "set attribute" requests that update non-security information such as access time or modify time are allowed even in NTFS-qtrees, and they do not delete the NT ACL.

6.2.3 Setting Permissions on Create

NFS creates in NTFS-qtrees are only allowed by the directory's owner. This is because an NT SID is required to handle NT ACL inheritance. An NFS request has no SID, but for a create request from a directory's owner, WAFL can use the owning SID from the directory's ACL and handle ACL inheritance according to normal NT rules.

In Mixed-qtrees, NFS create requests are handled according to normal UNIX rules.

7 UNIX to NT User Mapping

The first ACL release handles non-native NFS requests using permission mapping rather than user mapping, because of the cost and complexity of doing user mapping on every single NFS request.

However, we believe that with appropriate caching, the cost of UNIX to NT user mapping can be reduced to an acceptable level. User mapping requires the following steps:

- When an NFS request arrives, it contains a UID. The filer uses /etc/passwd (or NIS) to convert the UID into a UNIX username.
- The filer converts the UNIX username into an NT username using a mapping file. (If no mapping is specified, the filer uses the UNIX username.)
- The filer contacts the NT domain controller (DC) to determine the SID for the NT username. If there is no account for the name, the filer uses a default SID (set to "guest" by default).
- The filer contacts the DC to get the SIDs of all groups to which the user belongs.

With these mapping rules, the filer has a full set of NT authentication information, which allows it to validate NFS requests based on the NT ACL.

With NFS, there is no concept of a session, so the mapping must be done for each request. The steps above are too time consuming to perform on a per-request basis, so WAFL must cache the mappings. NT networking sessions may last for days or weeks, so it should be safe to cache UID-to-SID mappings for at least a few hours.

8 Other Issues

8.1 FAT versus NTFS

NT servers support both FAT filesystems and NTFS filesystems. FAT is the traditional DOS filesystem – it has no file-level security at all. The NTFS filesystem was designed for NT and supports NT ACLs.

Since NTFS-qtrees and Mixed-qtrees both support ACLs, they must be advertised to NT networking clients as "NTFS". (If they were advertised as "FAT", clients would assume that they had no ACLs, and would disable the interfaces for controlling ACLs.)

It is less obvious how to advertise UNIX-qtrees. One can make a case either way, as these two conflicting arguments show:

- Advertise as "FAT"

  UNIX-qtrees don't support ACLs, so advertising them as FAT sends a clear message to clients not to use ACLs. Advertising as NTFS would be confusing, since no ACLs are really present and any request to set ACLs will fail.

- Advertise as "NTFS"

  UNIX-qtrees support file level security, and advertising them as NTFS allows the filer to display the UNIX permissions using faked-up ACLs. Advertising as FAT would be confusing, because it would seem to imply that no file-level security is present.

In the end, we decided to advertise UNIX-qtrees as FAT, because this seems least likely to confuse Windows programs that absolutely must have ACLs.

Still, there are several situations in which it is useful to construct a fake ACL for an NT-style file, as described above in 5.2.1:

(1) In Mixed-qtrees

Mixed-qtrees contain both UNIX-style and NT-style files. To support ACLs they must be advertised as "NTFS", yet not all files in them contain ACLs.

(2) In NTFS-qtrees that originated as UNIX or Mixed-qtrees

A qtree's security style can be changed at any time, so a qtree that began as UNIX-qtree may later be converted to NTFS. In this case, it will clearly be advertised as "NTFS", but it may contain files without ACLs.

(3) In UNIX-qtrees accessed via an NTFS or Mixed Share.

The root of a filesystem may have NTFS or Mixed security, but it may contain a UNIX-qtree. In this case, the C$ (or root) share will be advertised as "NTFS", but a user can go down into the UNIX-qtree and then try to display an ACL.

8.2 Migration

For sites using old filers, migration to the NTFS security model is an important issue.

System administrators can update the security model for any qtree (including the root of the filesystem), using the **qtree** command. The syntax is:

qtree security *qtree* **[unix|ntfs|mixed]**

When a UNIX-qtree is converted to an NTFS-qtree, shares are advertised as "NTFS" instead of "FAT". The files themselves are not converted to NT-style files, so they behave as described in section 5, Handling Non-Native NT Requests. Of course, the owner of a file can set an ACL, converting the file to NT-style. Also, NetApp will ship a Windows utility to run through a tree and set a real ACL on each file based on its UNIX permissions. This is useful since the faked-up ACL doesn't show the exact permissions for a file.

When an NTFS-qtree is converted to a UNIX-qtree, shares are advertised as "FAT" instead of "NTFS", and any ACLs in the qtree are simply ignored. ACLs are not actually deleted – however – so if the qtree is converted back to NTFS, the ACLs will still be present. The best way to delete the ACLs is to write a script that runs as root and chowns each file to its existing owner. (Remember that doing a chown or chmod deletes the ACL on a file.)

8.3 Group Mapping

With user mapping, it shouldn't be necessary to map between NT and UNIX groups. When an NT user is mapped into a UNIX user, the filer also identifies the UNIX groups for that user, so access based on group rights will work correctly.

Unfortunately, this approach doesn't work for a user that isn't successfully mapped. Consider an NT user named "nt-john" who is a member of the group "nt-engineering". If "nt-john" successfully maps to the UNIX account "unix-john", then he'll get all group membership associated with "unix-john", presumably including "unix-engineering". On the other hand, if "nt-john" doesn't map to "unix-john", then he'll simply become UNIX "nobody", with no special group rights at all.

Thus, it might seem useful to explicitly map "nt-engineering" to "unix-engineering" so that group level access would be permitted even if user mapping fails.

On the other hand, maintaining a group-mapping file seems at least as hard as maintaining a user-mapping file. If NT and UNIX administration are sufficiently coordinated to map groups, why not just map users instead? It seems simpler to support just one kind of mapping rather than two.

8.4 Share Level ACLs

Share level ACLs are now based on NT SIDs, and they can be edited over the network using the NT Server Manager.

For backward compatibility, share level ACLs based on UNIX user names will continue to function, although they cannot be controlled via Server Manager.

8.5 POSIX ACLs

Although POSIX ACLs are not currently a requirement, we wanted to ensure that our design for NT ACLs did not preclude support for POSIX ACLs later. In fact, our integrated security model could easily be enhanced to allow UNIX-style files to have either POSIX ACLs or traditional UNIX permissions. Since there are subtle semantic differences between NT ACLs and POSIX ACLs, we would maintain the distinction between NT-style files and UNIX-style files, and we would continue to use user mapping to handle non-native requests.

POSIX ACLs would allow – but not require – some additional enhancements. For instance, a faked-up POSIX ACL could reflect the NT ACL more accurately than faked up UNIX permissions can.

9 Implementation

Earlier sections describe how WAFL uses NT ACLs when processing requests. This section focuses on two additional implementation issues. Section 9.1 describes how WAFL stores the ACL on disk, and section 9.2 describes the NT administrative protocols that the filer must support in order to correctly handle NT ACLs.

9.1 Storing NT ACLs

UNIX permissions are easy to store, because they have a small, fixed size. NT ACLs are more difficult to store, because they have a variable size, depending on the number of ACEs (Access Control Entries) they contain, and they can get quite large. NT currently restricts ACLs to 64KB, but that limit is arbitrary and could easily grow in the future.

WAFL's on-disk format uses 128-byte inodes to describe files, much like the Berkeley Fast Filesystem [Hitz94, McKusick84]. WAFL stores UNIX permissions in the inode itself, but NT ACLs obviously don't fit. Instead, files with NT ACLs have a pointer to a second inode, called a xinode (extended inode), that contains the ACL data. In essence, the ACL is being stored as a special hidden file.

To reduce storage overhead, WAFL shares xinodes whenever possible. If two files have exactly the same ACL, then WAFL points both files at the same xinode. The link count in the xinode tracks the number of references, just as it does for a regular file, and the xinode is deleted only when its link count drops to zero.

This technique produces incredible storage savings because large numbers of files have the same ACL. This makes sense if you consider how ACLs are set. At create time, files inherit the ACL from their parent directory, which means that the ACL will match an already existing one. And when ACLs are set manually, they are commonly applied to an entire subtree at once, so – again – a large number of files share the same ACL.

9.2 NT Administrative Protocols

To handle NT ACLs correctly, the NetApp filer must support a surprising number of NT administrative protocols. To understand why, consider what happens when an NT user pops up the ACL editor on a remote file.

First, the ACL editor contacts the filer server and requests the ACL. In order to display the ACL, the ACL editor must convert the SIDs in the ACL into human readable user names. One might expect the editor to contact the NT Domain Control (DC) directly to perform this conversion, but it does not. Instead it sends conversion requests to the file server. At first this seems surprising, but it makes sense when you consider that the client may be in a different NT domain than the file server it is talking to. In addition, an NT file server may define local users that are not known to the domain controller.

Editing an ACL generates even more requests. When creating a new ACL entry, the ACL editor must display a list of all possible users and groups, and – again – instead of contacting the DC directly, it requests this information from the file server.

Thus, to support ACLs, the NetApp filer must support a wide variety of NT administrative protocols, both as a server, in order to receive the appropriate requests, but also as a client so that it can forward requests on to the DC. In addition, in order to convince clients to talk with it, the filer must advertise itself as a full-fledged NT server, which requires it to speak even more NT administrative protocols.

The end result is that the NetApp filer supports many of the administrative interfaces that NT administrators expect in an NT file server. The filer is visible in the network neighborhood, and it can be managed using Server Manager and User Manager.

10 Bibliography

[Bach86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice-Hall, 1986

[Hitz94] Dave Hitz, James Lau, and Michael Malcolm, "File System Design for an NFS File Server Appliance." Winter USENIX Conference Proceedings, USENIX Association, Berkeley, CA, January, 1994.

[Hitz95] Dave Hitz. "An NFS File Server Appliance (TR-3001)." Network Appliance, Mountain View, California, March 1995.

[McKusick84] Marshall K. McKusick. "A Fast File System for UNIX." ACM Transactions on Computer Systems 2(3): 181-97, August 1984.

[Reichel93] Rob Reichel. "Inside Windows NT Security." Windows/DOS Developer's Journal, April 1993.

[Watson96] Andy Watson. "Multiprotocol Data Access: NFS, CIFS, and HTTP (TR-3014)." Network Appliance, Mountain View, California, December 1996.

Pluggable Authentication Modules for Windows NT

Naomaru Itoi and Peter Honeyman
itoi@eecs.umich.edu, honey@citi.umich.edu
Center for Information Technology Integration
University of Michigan
Ann Arbor

Abstract

To meet the challenge of integrating new methods and technologies into the Internet security framework, it is useful to hide low-level authentication mechanisms from application programmers, system administrators, and users, replacing them with abstractions at a higher level. The Pluggable Authentication Method approach popular in Linux, Solaris, and CDE offers one such abstraction.

To implement PAM in NT, we replaced the standard Graphical Identification and Authentication module with one that processes PAM tables. This provides security administrators with a flexible tool to plan and implement authentication policy across a wide range of computing platforms.

GINA is woven into the NT logon procedure, making it a difficult module to test and debug. Our PAM-based GINA eases this problem by allowing new authentication mechanisms to be replaced and tested without forcing a reboot.

1 Introduction

Security technologies are constantly evolving to meet the demands of Internet services. For example, network authentication protocols such as Kerberos [SNS88, KNT91], and Netware undergo periodic revision to meet new challenges. Similarly, the basis of secure authentication evolves, replacing password-based methods with ones that depend on smartcards or biometrics.

To meet the challenge of integrating new methods and technologies into the Internet security framework, it is useful to hide low-level authentication mechanisms (or AMs) from application programmers, system administrators, and users, replacing them with abstractions at a higher level. This allows the underlying mechanisms to be replaced as needed without changing APIs, documentation, or the "user experience."

Authentication protocols are used to protect Internet services. For example, the AFS file system is protected by Kerberos 4, and Netware protects its services, such as mailing and printing, with its own authentication mechanisms. Rapid evolution of authentication protocols forces users to learn these new authentication methods. In addition, in an environment with multiple Internet services, users are required to authenticate to every Internet service by typing passwords.

The Pluggable Authentication Module (PAM) framework provides a powerful abstraction for user identification and authentication. PAM defines a generic API for authentication, hiding underlying mechanisms. This provides for easy replacement of authentication components and offers an attractive solution to the "single sign-on" problem [SS95], obviating the need to invoke multiple commands or to type passwords multiple times.

PAM is implemented in Linux, Solaris, and the Common Desktop Environment (CDE), and is a *de facto* standard authentication framework. IETF is also conducting PAM standardization efforts.

In this paper, we describe an implementation of PAM in Windows NT. NT has a replaceable component for user authentication, called Graphical Identification and Authentication (GINA) [MS96]. We implement a subset of PAM, which we call NI_PAM, and integrate it into NT via the GINA. As it turns out, NI_PAM also greatly helps development and configuration of new GINA methods.

The remainder of this paper is organized as follows. In Section 2, we describe PAM. In Section 3, we explain the Windows NT authentication mechanism, especially GINA, and identify problems with GINA. Section 4 details our design of NI_PAM to improve GINA. We discuss results of the project and conclude in Section 5.

2 Pluggable Authentication Module

PAM is a framework that provides a generic way of authenticating users. PAM makes authentication components replaceable, defines a general API for authentication, and provides single sign-on for users. We discuss the role of each of these features.

2.1 Replaceable authentication modules

A system administrator may be required to replace an authentication mechanism and its components when new mechanism are developed, or because of a change in security policy. PAM provides a dynamically configurable ("pluggable") authentication mechanism by separating applications such as `login`, `ftp`, `telnet`, *etc.* from low-level authentication modules such as Kerberos, Netware, *etc.* PAM is controlled by a configuration table that defines the behavior of application programs. The configuration table is maintained by system administrators. Table 2.1 shows a simple example of the PAM configuration table.

Service	module_type	flag	module_path	Options
Login	Auth	required	pam_unix_auth.so	
Login	Auth	required	pam_kerberos.so	use_first_pass
Login	Auth	optional	pam_netware.so	use_mapped_pass

Table 2.1: PAM configuration table for `login`

The example, a configuration for a login application, uses three authentication mechanisms: UNIX `passwd`, Kerberos, and Netware. The `required` flag on a row means that a user must be authenticated by the listed mechanism to be accepted by the login program. The `optional` flag indicates that the system should attempt to authenticate the user to the listed mechanism, but the user can be accepted by login even if this authentication fails.

In the example, the login program requires a new user to be authenticated successfully with UNIX `passwd` and Kerberos. If she is authenticated by both, she is allowed to login. In addition, the login program attempts to authenticate the user to Netware, but this step is optional, so the user can login even if Netware authentication fails.

Changing the behavior of an application program is easily accomplished by modifying a PAM configuration table. For example, one can require login to use S/KEY authentication mechanism by adding a line to the configuration table:

```
login    auth    Required    pam_skey.so    Use_first_pass
```

Removing the requirement for Kerberos authentication is accomplished by removing the line for `pam_kerberos.so`. Observe that this modification can be made without (necessarily) writing, compiling, or installing any code.

2.2 API for application programs

Without PAM, applications must engage mechanisms specific to the various authentication methods, *e.g.*, by calling `krb5_get_in_tkt()`. With PAM, applications do not call authentication mechanisms directly. Instead, they call PAM API functions such as `pam_authenticate()`. Table 2.2 shows examples of the PAM API.

`pam_start()`	Initiate an authentication transaction
`pam_end()`	Terminate the authentication transaction
`pam_authenticate()`	Verify the identity of the current user
`pam_chtok()`	Change password

Table 2.2: PAM API examples

As a high-level abstraction, the PAM API hides details of low-level authentication mechanisms. This insulates applications written from the specifics of authentication mechanisms. Modification of the PAM configuration table does not affect application programs. For example, in Section 2.1, the login program is not changed when an authentication mechanism is added or removed.

2.3 Single Sign-On

A user is required to obtain credentials before using Internet services to prove she has the right to access the services. For example, the AFS file system allows access from a user with Kerberos 4 ticket. Credentials are obtained through authentication protocols. Computing infrastructures often support Internet services in multiple authentication domains, so a user is required to remember the various ways of authenticating, her many user names, and all her passwords. Users prefer to remember and use only one password and to obtain credentials for all resources by typing the password only once. This feature is called single sign-on [HAR95].

PAM provides single sign-on to users by sharing passwords among authentication mechanisms. In Figure 2.1, the `use_first_pass` option specified for Kerberos authentication directs PAM to use the password that the user provided. In this case, the user has the same UNIX and Kerberos password. The `use_mapped_pass` option in the Netware entry specifies that the password given by the user is not the Netware password, but is used to derive the Netware password. This might be necessary if Netware has special rules on password formats not enforced by other authentication methods, or if the Netware password is protected by the user-given password. In this way, PAM can give passwords to many authentication mechanisms, while requiring a user to type a password only once.

2.4 Example

To show the advantages of using PAM, we look at an example of how it simplifies login. Figure 2.3 depicts a system without PAM. The login program requires UNIX and Kerberos authentication. A method for authenticating in both UNIX and Kerberos must be hard coded into the login program. If a system administrator wants to enable access to Netware file service, she must add Netware authentication to the login by hard coding an authentication method for Netware into the login program. Worse yet, if a user wants to use `ftp` that accesses Netware, someone must then code this method into the `ftp` application, gaining little advantage from the earlier modifications to the login application.

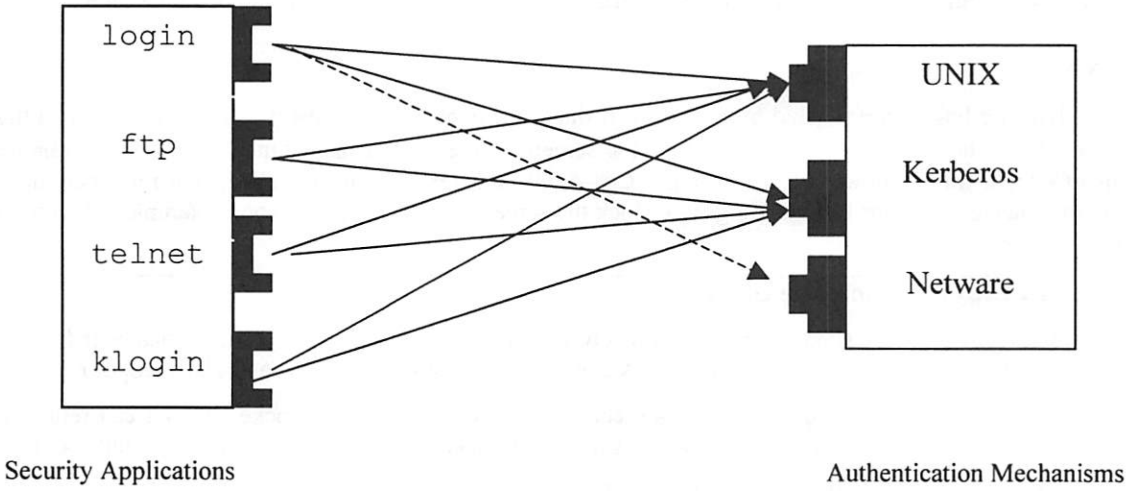


Figure 2.3: UNIX authentication without PAM

Figure 2.4 shows how PAM solves the problem.

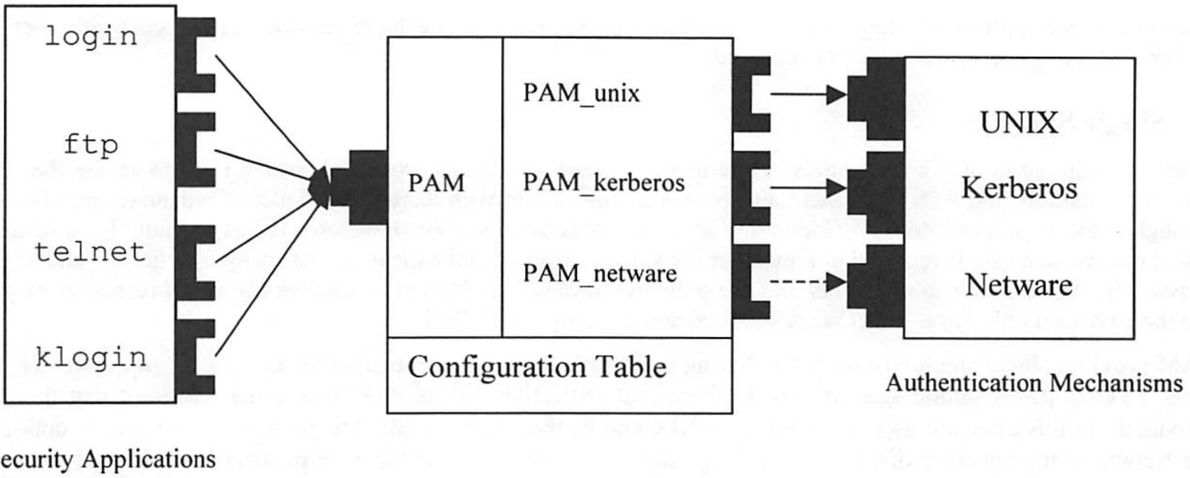


Figure 2.4: UNIX authentication with PAM

Here, applications are written with a single, generic PAM API call. Now the system administrator can add Netware authentication to the login program without writing any code in login or anywhere else. Instead, he adds one line in the PAM configuration table:

```
login    auth         required   pam_netware.so         use_first_pass
```

This example shows how PAM simplifies task of system administrators, application programmers, and users.

3 GINA: the NT authentication module

The component of NT responsible for interactive logon is called `Winlogon.exe`. To allow software developers to modify the authentication system of NT, a part of `Winlogon` called GINA is designed to be replaceable.

3.1 GINA

GINA is a dynamic link library loaded by `Winlogon` that is responsible for authenticating users. Authentication action is required when a user tries to logon, lock the screen, or logoff. At these times, `Winlogon` calls certain functions of GINA. In Windows NT, a user types Ctrl-Alt-Delete keys in unison, called SAS for secure attention sequence, to indicate she wants to logon, logoff, or lock the screen. Table 3.1 shows some examples of GINA APIs called by `Winlogon`.

`WlxInitialize()`	Initialize GINA.
`WlxLoggedOutSAS()`	Called when SAS is received and no user is logged on. This indicates that a logon attempt is made. GINA can return indicating the user is authenticated, or rejected.
`WlxWkstaLockedSAS`	Called when SAS is received and the workstation is locked. GINA can return indicating the workstation is to remain locked, or the workstation is to be unlocked.
`WlxLogoff()`	Called to notify GINA of a logoff operation on the computer.

Table 3.1: GINA API examples

To describe how GINA works, Figure 3.2 shows interaction among a user, Winlogon.exe, and GINA.dll.

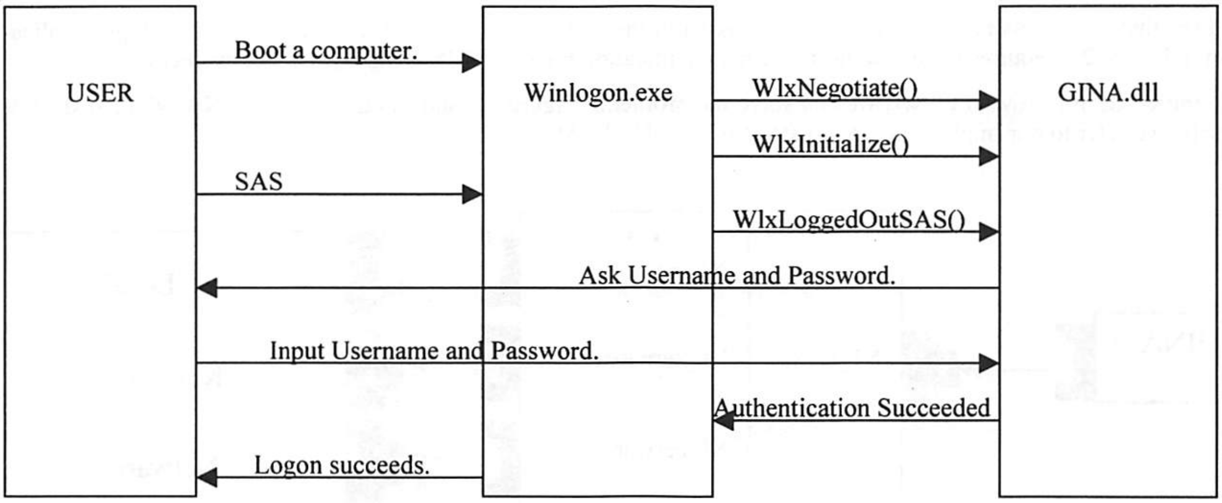


Figure 3.2: Interaction among a user, Winlogon, and GINA.

3.2 Problems with GINA

GINA is replaceable, which allows software developers to develop their own authentication mechanism. Many GINAs have been written for various authentication mechanisms, *e.g.* at the University of Michigan, we see frequent use of a Kerberos4-GINA, a Netware-GINA, *etc.* However, the structure of GINA poses some problems, which we now describe.

GINA is not PAM

While GINA is replaceable from Winlogon, GINA does not have the essential feature of PAM: authentication mechanisms are not replaceable in GINA. Figure 3.3 shows the standard Microsoft GINA, which authenticates a user to the NT local security system.

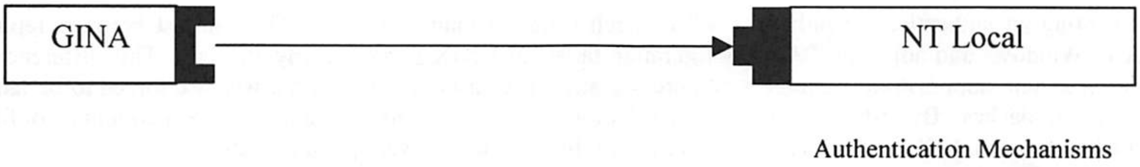


Figure 3.3: Microsoft GINA

Figure 3.4 shows Kerberos4-GINA, which authenticates a user to Kerberos4 as well as to Windows NT local.

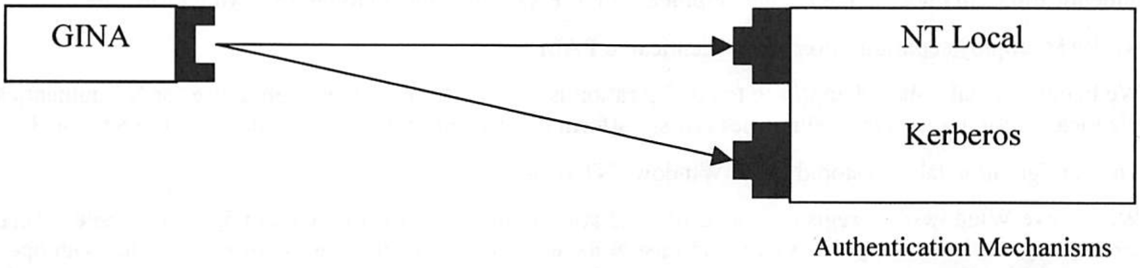


Figure 3.4: Kerberos4-GINA

We see that GINA has the same problems discussed in the context of UNIX login in Section 2. The login application in Figure 2.3 requires hand coding for each authentication method. GINA shares this inconvenience

By implementing PAM in GINA, we can solve the problem. Figure 3.5 shows our view of GINA with PAM. For clarity, we refer to our implementation of PAM in NT "NI_PAM".

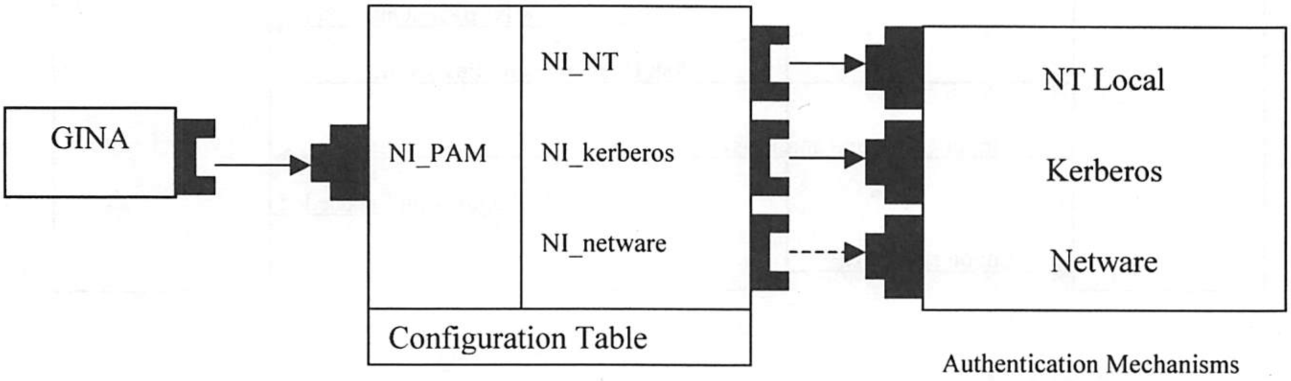


Figure 3.5: GINA with NI_PAM

GINA is hard to develop

Because UNIX applications run as user processes, applications such as `login` and `telnet` are relatively straightforward to develop and debug. GINA, on the other hand, is deeply integrated in NT logon, which makes it much harder to develop: a GINA developer must reboot the workstation for each new test of GINA. Furthermore, if the GINA does not work properly, the workstation hangs and cannot be restarted by usual methods. In addition, a GINA developer must use a debugger in an unusual way because GINA runs before logon is accomplished. In our experience, these two problems make GINA development very tedious. In contrast, NI_PAM eases GINA development by letting us test new authentication mechanisms without installing a new GINA, which obviates rebooting.

Lack of cross-platform security administration method

Implementing an authentication policy in NT is much different than in UNIX. The contrast between replacing GINA in Windows and adjusting PAM configuration tables in UNIX is particularly striking. This difference is a burden on system administrators charged with cross-platform security administration, who are forced to be familiar with both approaches. By installing NI_PAM in Windows NT, system administrators can use a common configuration method across UNIX and NT platforms. We are finding this to be a very popular feature.

4 Design of NI_PAM

As we saw in the previous section, implementing PAM in Windows NT aids the development and configuration of NT authentication. In undertaking the development of NI_PAM, we adhered to the following principles.

- NI_PAM employs configuration tables identical to PAM

  We believe the table-based approach to configuration is powerful and easy enough to use for NT authentication. Identical configuration tables simplifies cross-platform security administration, as described in Section 3.2.

- The configuration table is stored in the Windows NT registry.

  We believe Windows NT *registry* is a secure and appropriate place to store the configuration table. Access to the registry is allowed only for system administrators, and contains configuration information for both operating system and application software, *e.g.*, version numbers for software, search paths, names of DLLs called by the operating system, *etc.*

- Application programs other than GINA can call NI_PAM.

- NI_PAM assumes only one password for all authentication mechanisms.

We do not employ a password-mapping scheme in PAM. We discuss this further in Section 5.3.

4.1 NI_PAM components

Our implementation consists of three components: NI_PAM, NI_GINA, and the modules that implement authentication mechanisms, which we now describe.

NI_PAM

An application such as GINA calls NI_PAM.dll, the dynamic link library that implements PAM. On being called with a username, realm, and password, NI_PAM reads the configuration table in the registry and calls authentication methods with the combined arguments. NI_PAM receives a return value from each authentication module. Depending on the requirements of the PAM configuration table, NI_PAM returns to the application with success or failure.

For example, consider the (simplified) configuration table shown in Table 4.1.

Service	Flag	module_path
login	Required	ni_krb4.dll
login	Optional	ni_nw.dll

Table 4.1: A simple example of the NI_PAM configuration table.

When activated by Winlogon, GINA calls ni_authenticate with the username, realm, and password as arguments. By reading the configuration table, NI_PAM knows it must call the Kerberos-4 specific module, ni_krb4.dll, and the Netware specific module, ni_nw.dll. So NI_PAM calls ni_sm_authenticate(username, realm, password) of ni_krb4.dll and ni_nw.dll.

NI_GINA

NI_GINA, the GINA used with NI_PAM, is called by Winlogon and calls NI_PAM functions. For example, when a user tries to login, Winlogon calls `WlxLoggedOutSAS()` in NI_GINA. NI_GINA requests username, realm, and password from the user, and sends the information to NI_PAM, calling ni_authenticate with username, realm, and password as arguments. If NI_PAM returns NI_SUCCESS, NI_GINA does some post-authentication work, and returns SUCCESS to Winlogon.

Authentication Mechanism Specific Module

An Authentication Mechanism Specific Module (AMSM) implements mechanism specific authentication. For example, the AMSM for Kerberos-5, ni_krb5.dll, calls `krb5_get_in_tkt()` to get initial credentials from a Kerberos-5 KDC; the AMSM for Netware-4.0, ni_nw.dll), calls `NWDSLogin()`; *etc.*

Figure 4.2 shows the structure of the whole system that constitutes NI_PAM.

4.2 NI_PAM APIs

DLLs in NI_PAM structure export the following functions.

NI_GINA.dll Exports all functions defined in GINA [MS96].

NI_PAM.dll Defines the following data structure and exports the following functions.

```
// Data Structure
typedef struct _ni_struct {
       WCHAR  username[StrLen];          // user name
       WCHAR  domain[StrLen];            // domain name
       WCHAR  password[StrLen];          // password
       WCHAR  oldPassword[StrLen];       // old password, for change pw.
       WCHAR  newPassword[StrLen];       // new password, for change pw.
       INT          value;         // used for smartcard.
} NISTRUCT;
```

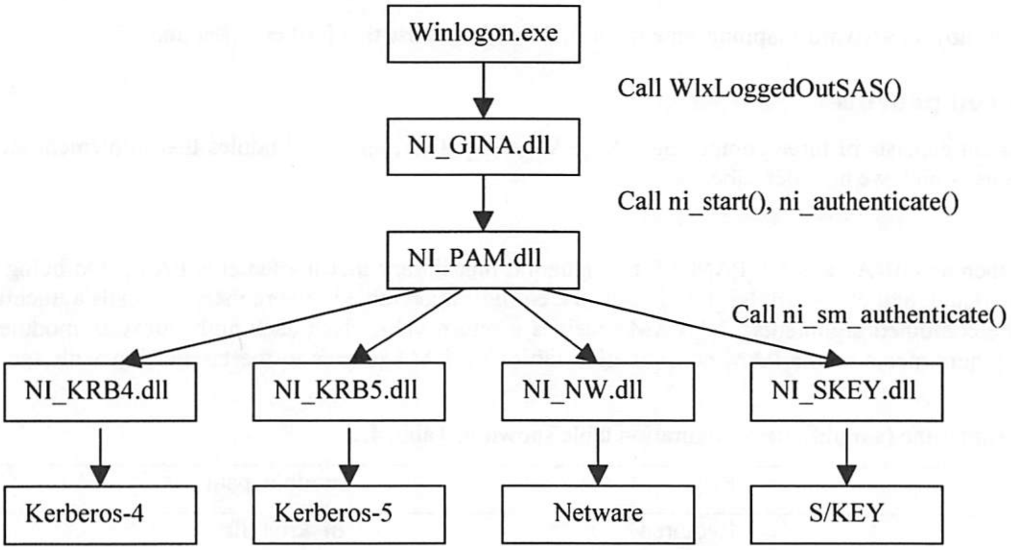


Figure 4.2: NI_PAM structure.

```
// prototypes
BOOL    WINAPI    ni_start(HANDLE    hWlx,    PVOID    pWinlogonFunctions);
        // Start NI_PAM transaction, read configuration tables.
BOOL    WINAPI    ni_obtainpass(NISTRUCT    *nistruct);
        // Attempt to obtain username and password from smartcard AMSM.
BOOL    WINAPI    ni_authenticate(NISTRUCT    *);
        // Attempt authentication to AM's, following the configuration tables.
BOOL    WINAPI    ni_logout();
        // Logout from AM's.
BOOL    WINAPI    ni_end();
        // END NI_PAM transaction.
```

AMSM Exports the following functions.

```
BOOL    WINAPI    ni_sm_authenticate(NISTRUCT    *niStruct);
        // Authenticate a user to an authentication mechanism.
BOOL    WINAPI    ni_sm_logout();
        // Logout from an authentication mechanism; destroy credentials.
BOOL    WINAPI    ni_sm_obtainpass(NISTRUCT    *nistruct);
// Returns username and password if this is smartcard AMSM.  Else, it returns NULL.
```

4.3 NI_PAM interaction

To show how the whole system works, we follow the function calls that occur in user authentication, given the structure in Figure 4.2 and the following configuration table.

Service	module_type	Flag	module_path
Login	auth	Required	ni_krb4.dll
Login	auth	Required	ni_krb5.dll
Login	auth	Optional	ni_nw.dll

- User issues SAS. Winlogon receives SAS. Winlogon.exe calls WlxLoggedOutSAS() in NI_GINA.
- NI_GINA.dll receives WlxLoggedOutSAS() and calls ni_start() in NI_PAM.
- NI_PAM.dll receives ni_start(), initializes global valuables, and reads the configuration table, and returns.
- NI_GINA demands (username, realm, password) from the user.
- NI_GINA.dll calls ni_authenticate() of NI_PAM.
- NI_PAM.dll receives ni_authenticate(), calls ni_sm_authenticate() of NI_KRB4.dll, NI_KRB5.dll, and NI_NW.dll.
- NI_KRB4.dll, NI_KRB5.dll, NI_NW.dll attempt to authenticate the user. They return NI_SUCCESS if they succeed otherwise NI_FAILURE.
- If NI_KRB4.dll and NI_KRB5.dll succeed, NI_PAM.dll returns NI_SUCCESS, otherwise NI_FAILURE
- If NI_PAM.dll returns NI_SUCCESS, NI_GINA.dll attempts local NT authentication. If the attempt succeeds, NI_GINA returns SUCCESS to Winlogon, else authentication fails, and NI_GINA returns FAILURE to Winlogon.
- Winlogon receives the result from NI_GINA. If success, user can logon. If not, she is rejected.

5 Results and Future Work

In this section we describe the status of the project, discuss the results, state future directions, and conclude.

5.1 NI_PAM status

NI_PAM is experimental software. Here is the status as of this writing.

- NI_PAM.dll is implemented and tested. We still run it in Debug mode, not yet in Release mode.
- NI_KRB4.dll and NI_NW.dll are implemented and tested. They support authentication and changing password.
- NI_KRB5.dll is implemented and tested. It supports authentication.
- NI_SC.dll (smartcard AMSM) is implemented and tested. Currently, this module stores passwords in the clear.
- NI_GINA.dll is implemented and tested, but needs more work before being deployed, including password changing, screen lock, static account, *etc.*

5.2 Results

Without NI_PAM, limitations in NT debugging facilities make a GINA very difficult to develop. It requires either rebooting every time it is tested or running "check-built" NT in one workstation with another NT workstation to monitor it. In contrast, with NI_PAM, separation of NI_GINA from the other parts involved in authentication (NI_PAM and AMSMs) lets us develop NI_PAM and AMSM's apart from NI_GINA. Indeed, we were able to implement NI_GINA by adding fewer than 20 lines of NI_PAM-specific code to an existing GINA. Thereafter, we were able to use NT's powerful debugging tools to develop NI_PAM and a handful of AMSMs.

Adding and modifying AMSMs in the future is straightforward. NT authentication system is now dynamically configured with identical way (*i.e.*, through the configuration table) with UNIX PAM.

NI_PAM provides single sign-on to a user. Once she is logged on to NT workstation, she has all credentials to access Internet services. Although we have not implemented any applications other than NI_GINA that uses NI_PAM, the NI_PAM API is generic and rich enough to be used by security-sensitive applications other than NI_GINA.

5.3 Discussion

We now consider of other ways of providing PAM functionality in Windows NT.

LSA

Local authentication is optional in UNIX-PAM, *i.e.* PAM can be configured to let a user login without a local account. This is an important feature in a large computing environment because it may not be feasible to store local account information for every potential user in every NT. The Local Security Authority, or LSA, is called from GINA and performs authentication to the NT local account. LSA ensures that the user has permission to access the system and provides a security access token. If we could modify or replace LSA, we could implement NI_PAM closer to UNIX-PAM in terms of local authentication. However, NI_PAM does not support this feature because unlike GINA, the LSA part of Windows NT cannot be modified or replaced

SSPI

SSPI is a generic interface for secure communication based on GSS-API. SSPI provides a higher abstraction to security protocols. It is similar to NI_PAM's goal. However, SSPI does not support user authentication. SSPI protocol starts with credential already obtained. Thus, NI_PAM and SSPI complement each other. NI_PAM provides a generic authentication method, and SSPI provides a generic secure communication.

5.4 Future Directions

Static Account

With tens of thousands of mobile users in our computing environment, it is not feasible to store account information for every potential user in every NT machine that she may use. In a heterogeneous computing environment with more than one operating system (*e.g.* UNIX and Windows NT), it is a real challenge to maintain the requisite multiple user account databases. An alternative is to allow a user to authenticate to a static account (*e.g.* "guest"). Here, the system keeps users' account and profiles information in server machines, and protects the database with a strong authentication mechanism (*e.g.* Kerberos-4). GINA-UM, developed primarily by Allan Bjorklund at the University of Michigan, implements this feature. After a user logs on to NT workstation, GINA UM retrieves the user account and profile from and AFS server, and uses that information to personalize the local machine.

Since static account is an important feature in a large computing environment, we are now looking for a way to implement it in NI_GINA, while preserving pluggability.

Password consistency

Kerberos has its own password database, as does Netware, *etc.* When a user attempts to change her password, some AMs may fail while others succeed. This makes it hard to preserve password consistency among multiple AMs.

For example, suppose a user wants to change password in both Kerberos and Netware. NI_PAM calls ni_chpass(old_password, new_password). Now suppose that after successfully changing her Kerberos password, the system suffers a network failure and that Netware fails to change the user's password entry. The system has inconsistent password databases – a new password for Kerberos, but an old password for Netware – and NI_PAM can no longer provide single sign-on.

Implementing password changing in NI_GINA causes a related problem. A user may change his password with other software, *e.g.* "User Manager", which does not employ NI_PAM, so it changes only the NT local password. That leaves other AM passwords unchanged, causing password inconsistency. To solve this problem, we are considering a DLL that is invoked on password change notification that uses NI_PAM functionality to change passwords in AMs according to NI_PAM configuration table.

Use of a single password for all AM's has its own problems.

- The security strength of the whole system is reduced to the strength of the weakest AM. A password compromised in the weakest AM yields the password to all AMs. We don't have enough experience with NT to have confidence that it protects passwords adequately. Software to obtain NT passwords is available in the Internet, *e.g.*, http://somarsoft.com/ntcrack.htm or http://www.ntsecurity.net/security/exploits.htm .
- Requirements for passwords are different among AMs. For example, some AM may require a password length more than eight characters, while another AM may require fewer than eight characters.

NI_PAM needs a way to use different passwords for each AM, while still preserving single sign-on. A common approach to this problem is to use an encrypted "key ring." In essence, a key ring acts as a secure password database, protected by an AM. The password entered by the user is the one that unseals the key ring. After authenticating to the AM guarding the database, the system can obtain passwords for all other AMs from the key ring. We are considering this approach, with an eye toward storing the key ring on a secure directory server.

Smartcard

Another way to store secrets securely is to store them on a secure token, such as a smartcard. A smartcard can improve password-based authentication: passwords are often short, easy-to-guess, and stored in users' heads; keys in smartcards, on the other hand, are long, randomly generated, and stored in tamper resistant hardware.

In addition to computer authentication, smartcards have many other potential applications, particularly in the realm of electronic commerce. We will eventually see all these functions integrated into smartcards. NT authentication with smartcards is one step toward these goals.

GINA details

Our focus is on authentication, yet GINA has other important roles to play. NI_GINA needs a better GUI, AFS support, and support for screen lock before we can deploy it.

5.5 Conclusion

We implemented a dynamically configurable authentication, a generic API for GINA and application programs, and a single sign-on in NT. Our approach was modeled after the success of UNIX PAM at solving these problems. Although we found some aspects of the NT authentication system hard to work with, NI_GINA made life a lot easier. We have implemented NI_GINA, which integrates PAM functionality into GINA, and are planning for campus deployment.

Acknowledgement

We thank Andy Adamson for advice and ideas. We are grateful to Allan Bjorklund for letting us use his GINA as a starting point.

References

[SS95] V. Samar and R. Schemers, "Unified Login with Pluggable Authentication Modules (PAM)," Request For Comments: 86.0, Open Software Foundation (October 1995).

[HAR95] Peter Honeyman, William A. Adamson and Jim Rees, "Joining Security Realms: A Single Login for Netware and Kerberos," *Proc. of Fifth USENIX UNIX Security Symp.*, Salt Lake City (June 1995).

[KNT91] John T. Kohl, B. Clifford Neuman, and Theodore Y. T'so, "The Evolution of the Kerberos Authentication System," *Distributed Open Systems*, pp. 78-94. IEEE Computer Society Press (1994).

[MS96] Microsoft, "Winlogon User Interface," Microsoft Win32 Software Development Kit for Microsoft Windows (1996)

[SNS88] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," *Proc. of the Winter 1988 USENIX Conf.* (February 1988).

Contact

We plan to make NI_PAM freely available. Please contact Naomaru Itoi at itoi@eecs.umich.edu. The project homepage is http://www-personal.engin.umich.edu/~itoi/

Montage - an ActiveX Container for Dynamic Interfaces

Gordon Woodhull
nodrog@ix.netcom.com
Stephen C. North
north@research.att.com
Information Visualization Research.
AT&T Laboratories
180 Park Ave.
Florham Park, NJ, USA

Abstract

Montage is a customizable, embeddable ActiveX container. Its client objects may be positioned dynamically by an external layout agent. *Montage* manages toolbars and user interface modes, integrating disparate components into a single, consistent interface. An important part of this task is supporting "group repositories" of related objects for data transfer operations such as cut-and-paste, drag-and-drop, save and load. *Montage* does not rely on large external libraries such as the Microsoft Foundation Classes, and thus is relatively lightweight. The prototypical *Montage* application is an embeddable display for dynamic networks (abstract graphs).

Introduction

An important trend in interactive computing is toward better integration of tools and services. ActiveX[1] is a protocol or set of interfaces for tool interconnection, enabling sharing of data and the user interface. ActiveX makes it possible, for example, to embed a live Excel spreadsheet in a WordPerfect document, or to place a third-party display widget on a Visual Basic canvas. ActiveX originated as an inter-client cut-and-paste protocol, but has grown to include many advanced services, including object naming, in-place editing, canvas event management (*e.g.* resize, drag-and-drop), toolbar and menu sharing, and loading and saving of persistent state. It employs the Component Object Model (COM) for communication between components. See Brockschmidt [2] for further background discussion on ActiveX and COM.

ActiveX is as difficult to program as it is feature-rich. It has dozens of interfaces, with hundreds of methods to call or implement. Although standard development tools simplify the design of contained objects (*controls*), we found insufficient support for designing new *containers* with a full complement of ActiveX features. We wrote *Montage* to provide this support in the form of a general container. We envisioned a small, efficient container object that would integrate with common desktop tools, having all aspects of its user interface and canvas layout determined by the components that run inside it.

The motivation for creating *Montage* arose from the Microsoft Windows port of *dotty* [5], a graph editor.[2] *Dotty* was written for Unix X Windows and is similar to other well-known Unix/X graph editors, such as GraphEd [7], Edge [9], and Graphlet [8]. *Dotty* was converted to MS Windows by re-coding its file selector, text-entry and other widgets in win32 graphics. This approach made *dotty* into an isolated island of X among MS Windows programs. Functioning almost identically to the Unix version, the port lacks even basic MS Windows compatibility such as access to native print drivers. Users would like a much higher level of tool integration, such as the ability to embed graph diagrams in text documents, or to incorporate multimedia controls in diagrams. Additionally, we encountered basic limitations in *dotty*'s architecture when our research in algorithms moved into dynamic on-line layouts – *dotty* assumes batch layout. All this suggested a fresh architectural approach.

In this context, we were interested to learn how much could be gained by wholesale adoption of ActiveX and win32 in a graph editor well adapted to MS Windows. As we developed a graph control, it became apparent that the layout and containment mechanisms could be cleanly separated. Our win32 graph editor became a general-purpose container, *Montage,* automated by a set of dynamic layout engines.

[1] Also known as Object Linking and Embedding or OLE.

[2] "Graph" always refers to an abstract graph or network in this paper.

A General ActiveX Container

In 1996, Microsoft drafted a new specification for what were then OLE Controls, OCX96, which among other things allowed for non-rectangular and transparent controls. Central to OCX96 was the definition of a *windowless control*. Before this time, all OLE objects used windows, which are overlapped rectangular regions that receive messages directly from the operating system. Windows were essential to early OLE because they meant that a visual region within one document could be owned by an application in an entirely different process. OCX96 defined new interfaces so that the container could ask the contained object mouse hit-testing and region opacity questions, and forward it the messages it would receive had it a window. Also, the contained object could ask the container for services, such as drawing handles and mouse capture, that normally require a window.

The facts of graph layout made OCX96 very appealing to us: nodes are usually non-rectangular, and it would be impractical to put opaque rectangles around edges. If *Dynagraph* were a windowless control container, both nodes and edges could be represented by ActiveX controls. But the first two versions of the Windows *Dynagraph* were written with the Microsoft Foundation Classes (MFC), which do not support containment of windowless controls. The Active Template Library (ATL) added efficient support for writing windowless controls in early 1997, but still there was no library support for containers.

Since it was clear that we would have to write any support for windowless controls from scratch, it seemed like an opportunity to abandon the cumbersome MFC entirely. Another aspect of OCX96 made this especially appealing: the facility for transparent controls would make it possible to extract all *Dynagraph*-specific behavior out of the containment functionality, leaving a general ActiveX container and a fairly simple graph application. The main design question became, "**If ActiveX defines a containment relation, what is the abstract data type for that relation?**" The *Montage* answer is to factor out all interaction, layout, coordinate transformation (zoom, etc.), and persistence, as for example a function for ordering objects is factored out of an ADT for dictionaries. What is left is a z-ordered list of positioned content objects at various levels of activation.

Montage is not the only solution to this problem. Dynamic HTML (which was developed concurrently) also moves layout policy out of the core container and into client programs. Yet browsers do not have persistence models: they are *presentation* oriented, not *document* oriented. Visual Basic and almost all other OLE applications with scripting languages also function as general containers, but developers may find themselves fighting with the layout and other peculiarities of the document data type. In contrast, the *Montage* document type is simply an ordered list of controls; we model the familiar OLE GUI in auxiliary controls and other components.

Automation

ActiveX Automation is the name for the invoking program functionality through public COM interfaces. An **automation object** implements and publishes interfaces; an **automator** (or "automation controller" or "client") calls methods in these interfaces. Wherever possible, *Montage* has been factored into components that communicate over such public interfaces. The layout engines, modes, and storage system are considered external clients, with no privileged access to *Montage* data. This design allows the easy replacement of most non-core parts of *Montage*, and also allows fine control over its functionality from any Automation-compatible language, including C++, Java, and in the future, Visual Basic and Web scripting languages such as JScript and VBScript. Through Automation, a *Montage* diagram can be connected to a dynamic process, for instance to show diagrams of a computer network. If the system can then react to diagram events, *Montage* becomes a "transparent" user interface to manipulate external objects.

Dynamic Layout Engines

A key requirement for *Montage* is to support dynamic network diagrams. The elements of network diagrams are nodes and edges. These are represented in *Montage* by ActiveX controls. For convenience we supply a "shape node" control specifically for graph diagrams, but other ActiveX documents or controls have equal status. Similarly, edges are ActiveX controls that draw generalized curves; we supply a simple non-interactive control to draw polylines and Bezier splines, but the component architecture leaves the door open for interactive edges, varying edge styles, etc. *Montage* supports both windowed and windowless controls. We use windowless controls for both edges and shape nodes; nodes having a naturally rectangular aspect, as do most ActiveX documents, may be either type.

Layout in *Montage* (and in all of ActiveX) centers on the **site** object, which represents the relationship between the container and contained object. The site's properties include position, a reference to the contained object/control, and ambient properties such as default colors and fonts. However, standard ActiveX does not define any interfaces to change these properties. In other words, container functionality can be divided into fetches, requests, and commands, but ActiveX only defines interfaces for the first two. Standard ActiveX does offer ways to give hints to containers, but policies about layout and activation are generally hidden.

In contrast, *Montage* pushes policy decisions out of the container. It defines interfaces for clients to directly change site properties. In this model, clients make hints or issue requests by treating them as events sent to sites. In the case of *Dynagraph*, to support layout control by a central server, the *Montage* architecture specifies that automators such as modes (described below) that need to change a diagram do this by generating events on the affected objects' sites rather than making changes directly. If a sink picks up an event, it changes the layout; otherwise an automator is free to make the change itself. In effect, the site has an **owner** (which may be the control itself) that interprets all requests made on the site, usually forwarding them to a layout engine (see Figure 1).

In our main application, the *Dynagraph* ActiveX control, the *Dynagraph* library maintains network diagrams. *Dynagraph* is a portable C library that defines an interface for incremental graph layout services. Clients can open and close diagrams and edit their contents by sending layout events to an engine. Events refer to operations on individual nodes and edges referenced by client-side descriptors, for example, "insert node *v* at (x,y)", "move edge *e* to $(x_0 ,y_0, x_1, y_{1.} x_{2.} y_2,\ldots)$" or "delete node *v*". To update live diagram displays, layout managers send events of the same type via callbacks. Note that a single request event may yield many diagram display update events, and it is also possible for an update request to be denied by a layout engine (say, if a request is inconsistent with diagram-specific constraints). Compound updates are handled by input event queues in layout engines. The library currently has managers for dynamic hierarchical layout [10], virtual physical ("spring") models, and incremental orthogonal layout.

In order to abstract the layout engines into a separate module, we created a COM wrapper for the *Dynagraph* library. The C structures are manipulable through COM interfaces, and the callbacks get broadcast through COM connection points. Then a separate module, called DGM (*Dynagraph for Montage*), manages the translation from *Montage* events to *Dynagraph* events, and from *Dynagraph* events to commands for *Montage*. This design helps ensure that neither is dependent on the details of the other.

Modes and Toolbars

Because *Montage* is a general control container, it has no built-in user interface. On its own it does not respond to mouse or keyboard events, so the only objects that can respond are controls activated within *Montage*. The user interface of a *Montage* application consists mainly of the controls plugged in as **modes** and as **toolbars**. Modes provide the main interface. These are transparent, windowless controls placed in front of all inactive objects but behind all active objects. Here they receive all mouse events that the active contained objects do not process on their own.

For example, a left mouse button drag consists of the Windows messages WM_LBUTTONDOWN, multiple WM_MOUSEMOVEs, and a WM_LBUTTONUP. If an active windowed control is under the mouse, Windows routes these messages directly to it, for instance selecting text in a Word document. Otherwise the messages go to the *Montage* window, which searches the Z order for the control under the mouse and forwards the messages. If there is no active object in front of it, the active mode will then receive the messages through the OCX96 interface IOleInPlaceObjectWindowless. If the mode interprets the drag as a move, it then asks *Montage* what control is below it at the coordinates of the WM_LBUTTONDOWN, and generates the event IMCCSiteOwner::Move on the site, or calls IMCCSite::put_Rect itself if there are no sinks for that event.

The design of modes as contained controls is a strategy to ensure that the *Montage* architecture is as open as possible. Even though two modes are compiled into the same dynamic link library (DLL) as *Montage*, all use only public interfaces to manipulate *Montage*. Accordingly, some of *Montage*'s internal calculations, such as "find control at point," are exposed as public methods, and *Montage* provides utility objects to simplify some of the common, cumbersome operations of modes, such as data transfer.

The two basic modes included with *Montage*, View and Edit, should be somewhat familiar from other ActiveX containers. **View mode** activates all contained objects so that they can be edited but does not allow any "structural" operations, such as moving, deleting, copying or pasting. **Edit mode** allows structural changes, as well as in-place activation with a double-click. Both modes should be applicable in many layout/UI applications. In fact, together they define a container with purely manual layout.

Unlike the generic View and Edit modes, the **Draw mode** of *Dynagraph* is entirely application-specific. Drawing (placing new objects) is a function of the user interface that will usually be specific to a class of applications, because the contained objects have various types that are presumably selected by various user-interface operations. In *Dynagraph* Draw mode, clicking on the canvas creates a new node, and dragging between two nodes creates a new edge.

In a typical application, multiple modes are active concurrently. ActiveX defines two levels of activation: **in-place** and **UI**. In-place activation only allows an object to receive mouse messages; a mouse message "falls through" any mode (or other windowless control) that does not catch it. UI Activation provides for toolbar, menu, and keyboard sharing, but unfortunately only defines negotiation between a container and *one* contained object. Presumably it would be much harder to negotiate between three or more graphic interfaces. But in the typical *Montage* case, we can assume that the concurrently active objects have been designed compatibly. So *Montage* defines a protocol for **Cooperative UI (coUI) Activation.** In coUI Activation:

- Keyboard messages fall through interested objects in the same way as mouse messages.
- Objects get a chance to add to the menu *Montage* negotiates with its container, and register to receive callbacks from their additions.
- The toolbar is exposed as a resource for all *Montage*-aware objects.

Additional services could ensure that the clients' uses of these resources do not collide.

To support toolbars, *Montage* can be activated "into" another window, typically the frame window of its container. Two auxiliary controls round out toolbar functionality. The **toolbar** control provides familiar grab handles around any ActiveX control, and can be floated. (To float, it removes the control and toolbar from the bar area, and re-activates the control into a new, independent floating window.) The **toolback** control provides the blank space on which toolbars can be arranged. Toolbars use the same layout architecture as graphs: moves initiated with the mouse get translated into requests to a **toolbar layout engine**, which ensures that toolbars don't pile up, while keeping them fairly close to where the user placed them.

Persistence and Data Transfer

Persistence is a key feature that must be implemented by ActiveX clients and servers, especially because data transfer actions such as cut-and-paste rely on the same mechanisms as the simpler save and load. Persistence of a collection of linked heterogeneous components requires some way to save the references between objects. The hierarchical persistence protocols provided by ActiveX make it possible for objects to create and initialize other objects, but there is no general way to connect objects not in a hierarchy. The problem may arise earlier: it is often only possible to create a live object by using a factory, yet the creator does not necessarily own (or know about) the factory.

Montage defines a new storage type called a **group** for this purpose. A group holds a set of named and anonymous objects, and supplies monikers for them. Monikers are standard ActiveX objects that can be "bound on" one object to get another; *Montage* uses "item monikers," which are the standard way to bind string names on a live object. A group's monikers, especially those identifying anonymous objects, are valid only within that group, so any group item that holds monikers must have them translated if the item is added to a new group. Because it would be inefficient and possibly incorrect to copy most live objects, the group model defines the **persistor**, a separate object that holds the monikers to reconnect an object on loading. The persistor gets copied and translated for each group, whereas its subject, the connected object, does not. A persistor may also hold a moniker to its subject's factory; in this way the persistor can be created without special knowledge although its subject cannot.

Persistors implement the interface IGroupItem, which allows them to control the way they are copied from group to group and to make sure any connected objects are copied as well. Since the persistors pull objects from group to group as necessary, drag-and-drop and cut-and-paste are fairly simple operations: the initiator of the transfer (usually a mode) creates a new group and pastes all objects to be transferred into the new group with IGroup::Paste. Then it saves the group to a storage object, and wraps the storage object in an ActiveX-compatible data object for the operating system. The destination of the transfer retrieves the group and pastes the objects into its own group to complete the transfer.

Data transfer between *Montage* and another application is also straightforward, as the ActiveX format negotiation protocol defines a standard for embedded objects. If items are dragged out of *Montage* and into another application, that application can embed them as a new instance of *Montage* because *Montage* supports the standard "embedded" format. Instead of transferring names (that the client will know nothing about), the client loads the data as a complete object. Similarly, if the user drags from a non-*Montage* object onto a *Montage* canvas, modes can accept this drop in the embedded object format as a new contained item, though they know nothing about the source application. (For example, Graph Draw Mode embeds an object as a new layout node.)

The *Dynagraph* Application

Figure 2 shows *Montage*, embedded in Microsoft Word, running *Dynagraph*. The session actually consists of three instances of *Montage*: the graph view, the toolbar area, and the node palette in the toolbar. Additional diagrams and views would involve more instances. Nodes may be added to a diagram by dragging items from the palette or another application to the canvas, or by clicking on the canvas to insert the object selected in the palette. Similarly, items can be added to the palette by dragging them from the canvas or from another application. It is satisfying that this capability arises naturally from the *Montage* architecture, as it took considerable programming effort to put similar features into *dotty*.

Figure 3 shows sample automator code that inserts a node into a graph. It demonstrates most of the common *Montage* operations: the manipulation of sites and layers, the use of transforms and connectors, and the activation model.

Unfortunately, the details of coding in COM may make the code somewhat cryptic to the uninitiated. The syntax of the code requires some explanation:

- All COM interfaces (which by convention start with the letter "I") extend the interface IUnknown, which provides methods for reference counting and querying for interfaces on the object. ATL's CComPtr and CComQIPtr template classes ("qi" is a macro based on CComQIPtr) abbreviate the use of these methods by automatically calling the IUnknown methods as necessary.
- All COM methods return error codes that should not be ignored, so the RUN macro echoes errors by returning them, effectively treating them as exceptions.
- Since COM only supports methods and not properties, it recommends that get and set methods be prefixed with "get_" and "put_" respectively. ("new_" is an analogous convention in *Montage* code for methods that create objects that share memory with their parents.)
- All classes that do not share memory have 128-bit IDs (whose constants start with "CLSID_"), and are created with the system call CoCreateInstance.

The code starts with the creation of a new site in the *Montage* container. Sites are the only way to refer to contained objects; they represent the work that *Montage* is doing to keep a contained object afloat. Next it places the site in the main layer, which is the lowest layer in the Z order in the standard configuration of *Montage*.

The call to CoCreateInstance instantiates the control which will provide the visual part of the node. This control may also come from a data transfer triggered by a drop or paste operation. (In Graph Draw Mode, the data source is the active selection in the palette.) Next the code tells the control its size through the standard ActiveX interface IOleObject. The control does not have to accept this size (*i.e.* SetExtent may return an error code), so code within the connector asks the size through GetExtent rather than trusting the argument value. This illustrates a recurrent design theme in ActiveX component design: just about every method call is a request that can be interpreted or ignored, a callback or future query works better than remembered state.

As described before, the *Dynagraph* application of *Montage* has three levels of objects. The next section of code creates both *Dynagraph* and DGM objects. The DGM object is made aware of the two sides it is mediating through the interfaces INeedOnePointer (of the Group model) and IObjectWithSite (of standard ActiveX). It is unfortunate that such non-specific methods must be used to set up the connector, but using generic interfaces makes persistence simpler. Specifically, INeedOnePointer makes it simple to define a generic persistor for the common case where the persisted object needs to be connected to exactly one other object when it is loaded, or in order to load. And IObjectWithSite makes it possible for the site persistor to tell the owners their property, without knowing what the site means to them. When it is told its site, the connector sets itself up as a sink of the IMCCSiteOwner event interface, so that it handles all layout events.

Now that the structure is in place, it is possible to actually begin the layout. The code here tells the *Dynagraph* node object directly what the position is; it could instead use the event interface on the site, but it can be pretty certain at this point that the *Dynagraph* node object will handle the event. First it must translate the coordinates, though: the engine expects canvas coordinates, expressed in units of .01mm (HIMETRIC), not the window coordinates the mode is working from. So this code fetches the transform object (another modular component) from *Montage*, and asks that to do the translation.

Finally the node can be shown. The code here uses the IMCCSiteOwner event interface on the site to issue a show request. We omit the code for clarity, but the function intend() enumerates the sinks of that interface on the site, and calls IMCCSiteOwner::Intend() on each.

Summary

Montage is a convenient framework for ActiveX container applications and controls. It does not replace ActiveX, but handles most of a container's busywork, while leaving actual interface policy decisions to the modes and layout engines. An interesting issue is whether *Montage* ought to have a more abstract interface, instead of one so closely tied to Microsoft Windows and COM. Whether this is worth much effort is moot without an alternative application embedding platform to target.

Our work list includes support for object linking (besides embedding), dispatch interfaces for compatibility with Visual Basic and scripting languages, DCOM, further development of coUI Activation, and replacement of some high-level C++ code (such as that in the example) with scripts. A more ambitious goal on the layout side is to support compound nodes and edges. Complications arise in managing layouts where edges are allowed to connect nodes at variable levels of nesting, as in Harel's Statecharts [6]. Because there is yet no automated Statechart drawing algorithm or tool, this is an interesting practical problem.

Acknowledgments

The authors thank Ted Kowalski from the AT&T Labs Software Practices and Technology Center for technical and financial support. We also thank Pierre Sierra for programming the first version of *Dynagraph*. We also gratefully acknowledge contributions by David Dobkin, Emden Gansner, Eleftherios Koutsofios and Phong Vo to the *Dynagraph* system.

Montage may be obtained under a non-commercial license at www.research.att.com/sw/tools/*Montage*.

References

[1] Barghouti, Naser, John Mocenigo and Wenke Lee. "Grappa: A Graph Package in Java", *Proc. Graph Drawing '97,* Lecture Notes in Computer Science vol. 1353, pp. 336-343, Berlin: Springer-Verlag, 1998. See also www.research.att.com/~john/Grappa

[2] Brockschmidt, Kraig. *Inside Ole, Second Edition.* Redmond: Microsoft Press, 1995.

[3] Cooper, Alan. *About Face: The Essentials of User Interface Design.* Foster City, CA: IDG Books, 1995.

[4] Ellson, John and Stephen North, "TclDG – a Tcl Extension for Dynamic Graphs", *Proc. 4th USENIX Tcl/Tk Workshop*, pp. 37-48, July 1996.

[5] Gansner, Emden and Stephen North. "An Open Graph Visualization System and its Application to Software Engineering," submitted, Nov. 1997.

[6] Harel, David. "On Visual Formalisms", Comm. ACM 31:5, pp. 514-530, May 1988.

[7] Himsolt, Michael. "GraphEd: An Interactive Graph Editor", *Proc. STACS '89*, Lecture Notes in Computer Science vol. 239, pp. 532-33.

[8] Himsolt, Michael. "The Graphlet System", *Proc. Graph Drawing '96,* Lecture Notes in Computer Science vol. 1190, pp. 233-240. See also www.uni-passau.de/~himsolt/Graphlet

[9] Newbery-Paulish, Frances and Walter F. Tichy. "EDGE: An Extendible Graph Editor"in *Software – Practice and Experience* 20:S1, pp. 64-88, 1990.

[10] North, Stephen. "Incremental Layouts in DynaDAG" in *Proc. Graph Drawing '95,* Lecture Notes in Computer Science vol. 1027, pp. 409-418. Berlin: Springer-Verlag, 1996.

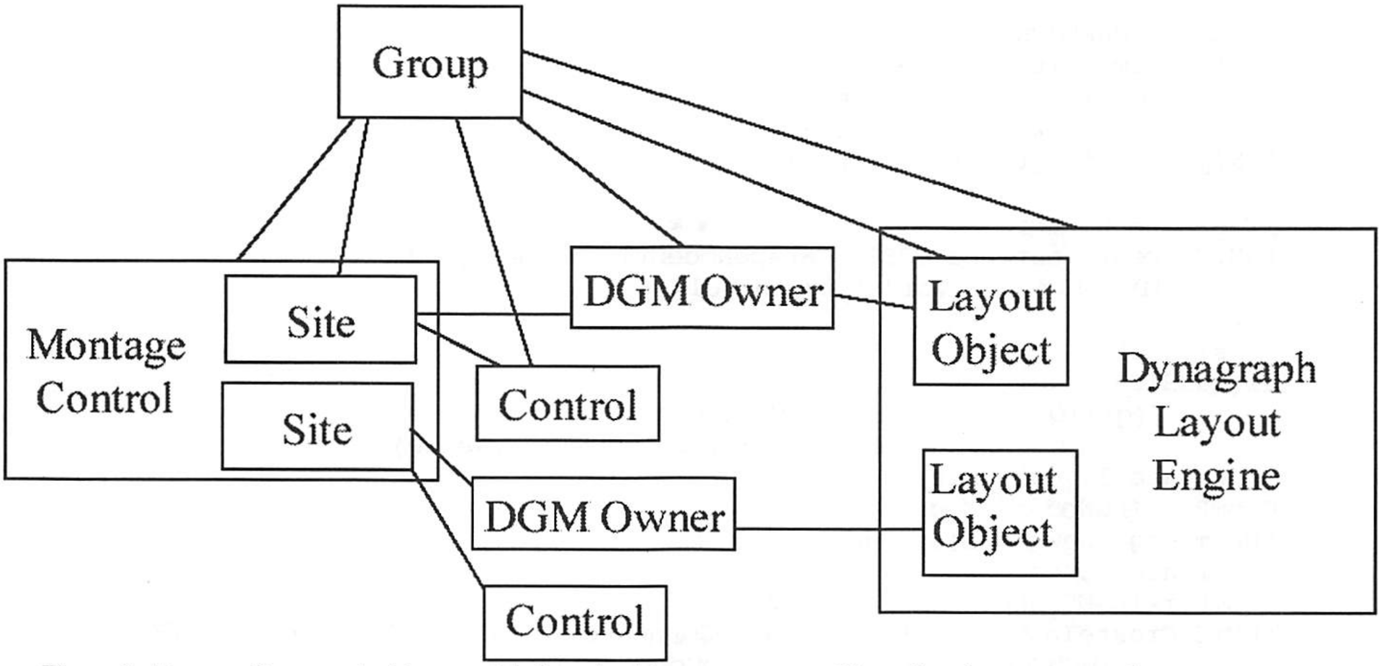
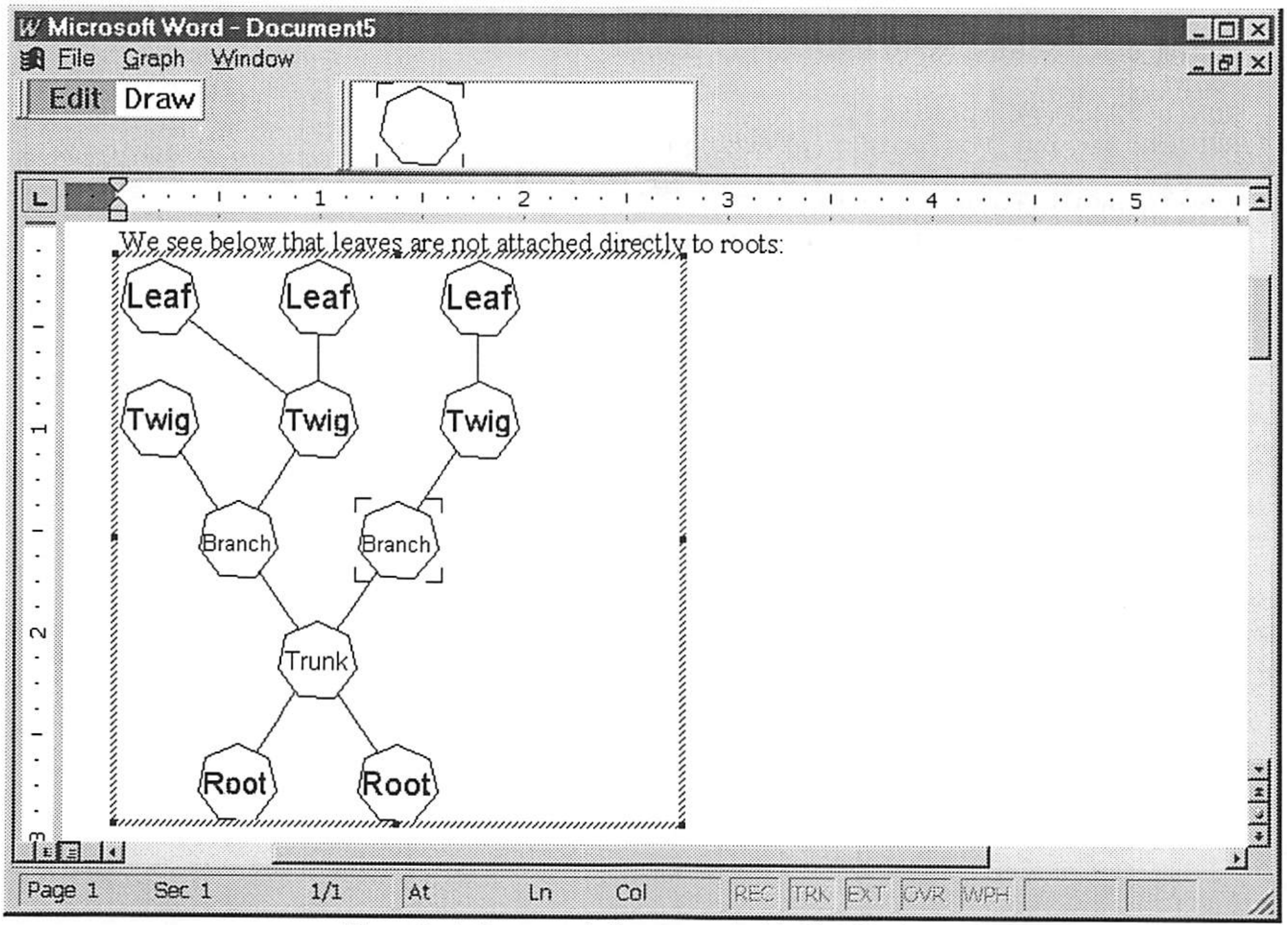


Figure 1: *Montage Dynagraph* object model. Contained boxes represent objects that share memory; lines represent communication over COM interfaces.



We see below that leaves are not attached directly to roots:

Figure 2: A *Dynagraph* diagram embedded in Word

```
CComPtr<IUnknown> pControl;
CComPtr<IMCCSite> pSite;
// create site in main layer
RUN(m_cont->new_Site(&pSite));
RUN(pSite->put_Layer(m_mainl));

// create control
RUN(CoCreateInstance(CLSID_CShapeNodeCtl,0,CLSCTX_ALL,
      IID_IUnknown,(void**)&pControl));

// size it.
if(hasSize)
      if(qi(IOleObject) oo = pControl)
            hr = oo->SetExtent(DVASPECT_CONTENT,&size);
// create layout node
CComPtr<IDGNode> pNode;
RUN(m_eng->new_Node(&pNode));
// Create connector & init.
CComPtr<IDGMConnector> pConnect;
RUN(CoCreateInstance(CLSID_DGMNodeConnector,NULL,CLSCTX_INPROC_SERVER,
      IID_IDGMConnector,(void **)&pConnect));
RUN(qi(INeedOnePointer)(pConnect)->TakePointer(pNode));
RUN(qi(IObjectWithSite)(pConnect)->SetSite(site));
// set position.
CComPtr<IMCCTransform> t;
RUN(m_cont->get_Transform(&t));
POINTL ptVirt;
RUN(t->W2CP(pt,&ptVirt));
RUN(pNode->put_Pos(pointf(ptVirt)));
// prob. just connector is sinked by now, but you never know.
RUN(intend(site,MCCS_SHOWN));
```

Figure 3: The code required to insert a *Dynagraph* node

SecureShare: Safe UNIX/Windows File Sharing through Multiprotocol Locking

Andrea J. Borr
aborr@hummosa.com

Abstract

As mixed UNIX/Windows environments become more common, safe file sharing among the NFS and CIFS clients becomes a significant problem. In this paper, we describe SecureShare, a multiprotocol file sharing technology that resolves the mismatch between NFS/NLM and CIFS file locking protocols, provides coherent caching and enables multiprotocol change notification. SecureShare achieves the goal through a uniform lock-mode model and multiprotocol oplock management. This paper presents the main features of SecureShare and discusses the rationale behind its design.

1. Introduction

Today, mixed UNIX/Windows environments are becoming increasingly common. Many commercial and academic institutes typically employ a mixed network of UNIX clients using the Network File System [3] (optionally with Network Lock Manager [5]) and Windows clients using either the Common Internet File System [6] or "(PC)NFS" [7]. Sharing files across the different systems is highly desirable and commonly done. However, correct concurrent read/write accesses from the different types of clients present a significant problem, because of the mismatch between NFS/NLM locking protocols and CIFS locking protocols.

Uncoordinated concurrent read/write accesses to files and directories can result in application failures or file data integrity problems. Examples of such problems are (1) readers receiving "stale" data (i.e. data currently in the process of being updated by another application), (2) writers overwriting each other's updates, and (3) applications having their in-use files deleted "out from under" them. Locking is typically used to coordinate concurrent accesses to files and directories, and prevents such problems from occurring.

Unfortunately, Windows and UNIX differ considerably in how they allow applications to access files and data while controlling the concurrency issues that arise in multi-user, multi-application environments. Windows implements a robust and complete set of file-open, data access and locking paradigms. UNIX, by contrast, typically implements only a basic data access facility, ignoring for the most part issues of concurrency and file sharing altogether. These differences in the handling of locking and file sharing issues carry over into the designs of the NFS and CIFS protocols used by UNIX and Windows, respectively, for remote data access. For example, CIFS transmits file open requests and byte-range lock requests to the file server, whereas NFS does not.

Correct interoperability of CIFS and NFS/NLM is impeded by the fact that CIFS and NFS/NLM have different and incompatible semantics for file-locking, file-open, and file sharing. The two principal interoperability problems are (1) CIFS mandatory locking vs. NFS/NLM advisory locking; (2) CIFS hierarchical locking vs. NFS/NLM non-hierarchical locking.

CIFS requires that the file server and all of its clients conform to the mandatory locking model. In the mandatory model, the file opener's (alternatively, the byte-range lock owner's) exclusivity of access to the opened file (locked byte-range) is enforced by the file server at the system level.

By contrast, NFS/NLM lacks file-open functionality, and provides only advisory locking. In the advisory model, system-level enforcement is replaced by application-level compliance. In this environment, each of a set of applications that read and write shared data depends for its correct functioning on global compliance with advisory locking rules.

In a mixed CIFS and NFS/NLM network, the incompatibility of the mandatory and advisory models poses a risk to data integrity. Windows-based applications depend on the stricter CIFS mandatory model for their correct functioning and for the integrity of their data. These applications may fail in the face of NFS accesses that write, remove, or otherwise corrupt in-use Windows files in a manner that violates the stricter CIFS mandatory model, but is permitted under the looser NFS/NLM advisory model.

The second problem impeding correct interoperability of CIFS and NFS/NLM is the fact CIFS expects all clients to conform to a hierarchical locking model. Correct application functioning and data integrity under CIFS rest on the assumption that a client first acquires a file-lock (i.e. by opening a file) before requesting a byte-range lock within the file. The NFS/NLM protocol, on the other hand, has no concept of a locking hierarchy. NFS/NLM has provision for acquiring a byte-range lock on a file, yet lacks provision for pre-acquiring a file-lock on the file by opening the file.

Network Appliance's solution to these problems is SecureShare [1], a multiprotocol file sharing technology. SecureShare is integrated into the *Data ONTAP™* microkernel, Network Appliance's operating system for its proprietary file server appliance, or *filer* [2]. SecureShare provides correct semantics for file sharing, opportunistic locking, byte-range locking, coherent caching, and change notification in a mixed network of UNIX clients using NFS, optionally with NLM, and Windows clients using either CIFS or (PC)NFS.

The key features of SecureShare are:

- uniform lock mode and multiprotocol lock enforcement
- multiprotocol oplock management
- multiprotocol change-notify.

In implementing multiprotocol data integrity, SecureShare reconciles the different and incompatible locking and file-open semantics utilized by CIFS and NFS/NLM clients. In implementing multiprotocol oplock management, SecureShare supports standard CIFS oplocks, while at the same time making oplocked data available to NFS-based clients through multiprotocol oplock break. In implementing multiprotocol change-notify, SecureShare supports standard CIFS change-notify, while extending the change-notification service such that it covers changes due to NFS in addition to covering changes due to CIFS.

SecureShare implements a uniform set of file-locking semantics for the multiprotocol environment. To achieve consistent multiprotocol locking and file-open semantics, SecureShare manages all locks according to a uniform lock management model. In this model, an expedient described in Section 2.2 is used to overcome the interoperability problem posed by the hierarchical locking conformance mismatch between CIFS and NFS/NLM. Uniform lock management also solves the interoperability problem posed by the mandatory vs. advisory mismatch. Lock enforcement under SecureShare depends on the lock type, the protocol that set the lock, and the protocol performing a file access. Whole-file locks (representing file-opens and oplocks) are enforced uniformly on the mandatory model for both CIFS and NFS file accesses. Byte-range locks may be enforced either on the mandatory or the advisory model, depending on which protocol created the lock and which protocol is performing the read or write. In particular, NLM byte-range locks are treated as advisory with respect to NFS reads and writes, in accordance with NFS/NLM protocol standards.

The cornerstone of SecureShare is a multiprotocol lock manager. The lock manager coordinates and manages — in a unified set of kernel-space data structures — the lock types needed for multiprotocol file-open and locking support. Since it is integrated into the *Data ONTAP* kernel, the lock manager is able to validate that reads and writes of files and directories do not violate locks. SecureShare enforces CIFS locks and file-open semantics at the system level. By contrast, in a UNIX-based CIFS implementation such as Samba [9], Windows file-open information is maintained by a module that is disjoint from — and has no interaction with — the module that implements UNIX and NLM locking functionality. Moreover, because data access functions under UNIX do not normally check for lock conflicts, there is nothing to stop local or NFS-based UNIX users and applications from accessing, corrupting, or even removing "locked" Windows files and data. Thus, it is possible in such an environment for UNIX users or NFS clients to write, remove, or move files that CIFS-based Windows applications are holding open and actively accessing.

SecureShare implements a multiprotocol extrapolation of the Windows coherent caching and networking performance optimization known as "opportunistic locks" (*oplocks*) [6]. By extrapolating oplocks to the multiprotocol environment, SecureShare allows CIFS clients to safely reap oplocks' performance benefits of aggressive client-side caching without causing Windows applications to suffer the effects of potential data corruption due to uncoordinated NFS accesses. By allowing non-CIFS requests to *break* (i.e. revoke) CIFS oplocks, SecureShare ensures that oplocked file data remains available to NFS-based applications and users, while simultaneously protecting the integrity and cache coherence of that data.

The remainder of this paper is organized as follows. Section 2 discusses file and data locking paradigms in the CIFS, UNIX/NFS, and (PC)NFS environments, and presents the uniform lock-mode model in SecureShare. Section 3 discusses multiprotocol lock enforcement. Section 4 discusses the CIFS opportunistic locking model and SecureShare's multiprotocol implementation. Section 5 discusses the CIFS *change-notify* feature and SecureShare's multiprotocol implementation. Finally , Section 6 summarizes.

2. Data Locking Paradigms

A lock on a data element grants to the lock owner a degree of exclusivity of access represented by its *lock-*

mode. The lock *type* specifies the span of data covered by the lock. In the case of a file server, lock types are categorized as a two-level hierarchy:

- File-locks (pertain to whole files)
- Byte-range locks (also called "record" locks)

2.1. Lock Models in CIFS and NFS

In the Windows (and CIFS protocol) model, a file-lock is obtained as a side effect of a *file-open* operation. The file opener specifies the *access-mode* it desires and the *deny-mode* that it desires to impose on other openers of the same file. In granting a new file-open request, the CIFS file server validates that the access- and deny-mode requested by the new opener do not conflict with the access- and deny-mode granted to pre-existing file openers. It then creates a *file-lock* that represents the access- and deny-mode granted to the new file-open.

In Windows (and CIFS), the application (CIFS client) must perform a file-open operation before attempting to read or write data contained in the file. After the open request has been granted, the CIFS file server validates that reads or writes requested on the open file comply with the access-mode granted by the file-open. CIFS expects *all clients* to conform to a hierarchical locking model. Correct application functioning and data integrity under CIFS rest on the assumption that a client first acquires a file-lock (i.e. by opening a file) before requesting a byte-range lock within the file.

By contrast, the UNIX file-open API does not support the concept of a *deny-mode* that it desires to impose on other openers of the same file. Lacking the concept of a deny-mode, the UNIX opener lacks a means of specifying a file-lock, since a lock by its very nature embodies the specification of a degree of exclusivity of access. Furthermore — because it was designed to be stateless [4] and file-opens are inherently stateful — NFS does not transmit even the *access-mode* declared by the UNIX file opener to the file server. Thus, the UNIX/NFS client cannot perform an operation that results in the acquisition of a file-lock at the server. Nor can it pre-declare to the server its intent to issue file reads or writes. Since the UNIX/NFS client can issue an NLM byte-range lock request for a file (see below) without holding a file-lock for the file, NFS/NLM is seen to violate the CIFS hierarchical locking model. The implications of this violation for SecureShare's multiprotocol lock enforcement are seen in Section 3.

(PC)NFS implementations for DOS and Windows emulate CIFS file-open functionality through use of NLM "share" locks, added to the NLM protocol for the specific purpose of supporting these clients. Called NLM "file-locks" in this paper, they are equivalent to CIFS file-locks. (This functionality is not normally exposed to the UNIX application environment, and so most UNIX users are unaware of its presence). The (PC)NFS client requesting an NLM file-lock specifies an *access-mode* and a *deny-mode*. In granting a new NLM file-lock, the server validates that the access- and deny-mode requested by the new request do not conflict with the access- and deny-mode granted to any pre-existing file-lock on the same file. The server then creates an NLM file-lock that emulates the presence of a new "file-open."

Byte-range locks are used to restrict other applications' access to sections of an open file, usually while the holder of the byte-range lock is intending to read or write the locked section. Byte-range locks can be either *read* locks or *write* locks. Depending on whether enforcement is advisory or mandatory, read locks either induce or enforce that other applications refrain from *writing* to the specified byte-range. Similarly, write locks either induce or enforce that other applications refrain from reading or writing the specified byte-range.

Both Windows and UNIX provide byte-range locking functionality. The Windows byte-range lock API is directly propagated via the CIFS protocol to the CIFS file server, where the resultant lock is enforced in accordance with the mandatory model. Since lock management is inherently stateful, NFS does not transmit UNIX byte-range lock requests to the server. Because the utility of locking in a file-sharing environment was recognized, however, the adjunct protocol NLM was defined to transmit UNIX byte-range lock requests to the server. NFS treats NLM locks as merely advisory, however.

2.2. The Uniform Lock-Mode Model

SecureShare implements a *uniform lock-mode* encompassing both file-locks and byte-range locks. The lock-mode combines a specification of the *access-mode* the lock requester desires (*Read*, *Write*, *Read-Write*), and the *deny-mode* it desires to impose on other clients concurrently attempting to access the same data (*Deny-None*, *Deny-Read*, *Deny-Write*, *Deny-All*). Some examples of uniform lock-modes are *Read/Deny-Write* and *Read-Write/Deny-All*.

When SecureShare creates a file-lock to represent a CIFS or (PC)NFS file-open, it derives a uniform lock-mode for the file-lock that combines the requested *access-mode* and *deny-mode*. In granting the new request, SecureShare validates that the derived lock-mode does not conflict with the lock-mode granted to any pre-existing file-lock on the same file.

CIFS or NLM byte-range locks are assigned uniform lock-modes as follows. In the case of a *write-lock* (also called an *exclusive* byte-range lock), the locked byte-range is *Read-Write* for the holder of the lock and *Deny-All* for others. Thus, the uniform lock-mode is *Read-Write/Deny-All* (RW/DA), and the lock is effectively "exclusive." A write-lock is grantable to only one "writer" of the byte-range at a time. In the case of a *read-lock* (also called a *non-exclusive* byte-range lock), the locked byte-range is *Read* for the holder of the lock and *Deny-Write* for others. Thus, the uniform lock-mode is *Read/Deny-Write* (R/DW), and the lock is "sharable." A read-lock is concurrently grantable to multiple "readers" of the byte-range, but is incompatible with a "writer" attempting to obtain a write-lock on the byte-range.

UNIX/NFS clients issue NLM byte-range locks without holding file-locks. This presents a problem for strict hierarchical lock enforcement, in which byte-range locks are not directly comparable to file-locks. SecureShare overcomes this problem through an expedient that allows a direct compatibility check of a file-lock against an NLM byte-range lock (see Table 2 at end of paper). Such comparisons, of course, violate the principles of hierarchical locking. The anomaly is circumvented by considering the NLM byte-range lock to have a deny-mode of *Deny-None* for purposes of comparison with a file-lock. After all, a write-lock's deny-mode — *Deny-All* — applies only to the byte range, not to the whole file. On the other hand, the write-lock's access-mode — *Read-Write* — signals the lock owner's intention to write to the file. In fact, since NFS provides no means of declaring a file-open time access-mode, the NFS/NLM client's only means of pre-declaring an intention to write the file is to acquire an NLM write-lock on one or more of the file's byte-ranges.

3. Multiprotocol Lock Enforcement

This section describes how SecureShare manages locks in a multiprotocol environment. It describes SecureShare's multiprotocol lock enforcement policies for CIFS file-locks, CIFS byte-range locks, and NLM byte-range locks. It highlights the risks to data integrity that could arise in the multiprotocol environment if locking were not managed in the described way. In describing various cases of multiprotocol lock enforcement, it points out scenarios in which oplock break protocol (discussed in Section 4) is triggered.

3.1. CIFS File-Lock Enforcement

SecureShare fully enforces Windows mandatory file locking across the CIFS, (PC)NFS and UNIX/NFS data access environments. When a Windows client attempts to open a file residing on a Network Appliance filer, SecureShare tests whether the open request can be granted based on the following criteria:

If other CIFS or (PC)NFS clients already have the file open, a "file-lock compatibility check" is performed to determine whether the access- and deny-mode of the new open are in conflict with access- and deny-mode(s) already granted to pre-existing opens of the same file. Computation of the compatibility check utilizes Table 1 (at end of paper), known in database management locking terminology as a *lock compatibility matrix* [8]. First, the access- and deny-mode requested by the new open are combined into a *uniform lock-mode*. Next, if there are multiple pre-existing opens, a *lock conversion matrix* (as defined in [8]) is used iteratively to combine the lock-modes implied by the access- and deny-mode(s) of pre-existing opens into a *cumulative lock-mode*. Then, Table 1 is used to test the compatibility of the lock-mode requested by the new open with the cumulative lock-mode of pre-existing opens. If the new lock-mode is not compatible, then the file-open request fails with a "sharing violation" error.

Suppose that — prior to the filer's receipt of the CIFS file-open request — a UNIX/NFS client has obtained an NLM byte-range lock on the file being opened. In this case, SecureShare uses Table 2 (at end of paper) to check whether the *deny-mode* component of the file lock-mode of the new open is compatible with the access-mode component (*Read* or *Read-Write*) of the lock-mode of the NLM byte-range lock. Note that this check — which violates the locking hierarchy — compares a file-lock with a byte-range lock. The rationale for this check is that — in the absence of NFS file-open functionality — the only way for a UNIX client to pre-declare its intention to make read (write) accesses to a file is by acquiring a read-lock (write-lock) on one or more byte-ranges.

Once one or more Windows clients have successfully opened a file on a Network Appliance filer, SecureShare coordinates subsequent NFS access to the open file in accordance with the deny-mode(s) granted to the openers.

SecureShare will reject any attempt by an NFS client (UNIX or Windows) to delete or rename a file that is being held open (after a possible oplock break sequence) by a Windows client. Note, however, that CIFS provides protocol for specifying a file open operation (i.e. *NTCreateX* with *share delete* mode) that permits *another CIFS client* to delete or rename a file (or directory) that is being held open.

SecureShare will reject any attempt by an NFS client to write to a file that is being held open (after a possible oplock break sequence) by a Windows client with a deny-mode of *Deny-Write* or *Deny-All.*

SecureShare is able to manage NFS file access and file system manipulation functions at the system level. Thus, it can deny NFS operations — such as writes, removes, renames, or moves — that could overwrite and/or corrupt a Windows-open file, violate a CIFS byte-range lock, or delete the file "out from under" a Windows application.

By contrast, because data access functions under UNIX do not normally check for lock conflicts, Windows mandatory locks are typically not enforced in UNIX-based CIFS implementations such as Samba. In such an environment, there is nothing to stop local or NFS-based UNIX users and applications from accessing, corrupting, or even removing "locked" Windows files and data.

On the other hand, SecureShare *does* allow UNIX/NFS clients to obtain read-only access to files that are actively being held open by Windows applications, even if those files were opened with deny-mode *Deny-Read* or *Deny-All.* Note that we are referring here to *actively open* files, as distinguished from files that temporarily *appear* to be open due to breakable stale batch oplocks (discussed in Section 4). This design decision was consciously made with consideration to practicality over technical "purity." The ability of a UNIX/NFS client to read files that are actively open by Windows applications cannot corrupt the file data from the CIFS point of view, and the NFS client's perceived cache coherence is no worse than is normal under NFS. However, if SecureShare were to deny all UNIX/NFS clients' attempts to read files that are open under Windows, this could generate an unacceptably large number of seemingly "inexplicable" errors for UNIX/NFS clients in the multiprotocol environment. This design decision is compatible with SecureShare's goal of protecting Windows data from corruption by NFS write accesses. It is arguably preferable to allow an NFS read request to return "dirty" data than to fail the request with a seemingly inexplicable "access violation" error.

3.2. CIFS Byte-Range Lock Enforcement

Since a file must be opened via CIFS before a byte-range lock request can occur, a CIFS byte-range lock request never occasions an oplock break (it would have occurred at file open time). Enforcement of CIFS byte-range locks in a multiprotocol environment occurs as follows:

- If another CIFS client already holds a byte-range lock that conflicts with the new CIFS byte-range lock being requested, then the new lock request will be denied.
- If a UNIX/NFS client or a (PC)NFS client already holds an NLM byte-range lock that conflicts with the new CIFS byte-range lock, then the new lock request will be denied.
- If neither of the above violations is detected, then the new CIFS lock request will be granted.

SecureShare treats CIFS byte-range locks as mandatory with respect to NFS file access functions, assuring that NFS reads and writes do not violate CIFS byte-range locks. By contrast, a UNIX-based multiprotocol file server such as Samba has no easy way to prevent violation of CIFS locks by NFS file access functions. Thus, it would be possible — due to the advisory nature of UNIX/NFS locking — for UNIX users and applications to write data to the CIFS "locked" byte-ranges of files.

3.3. UNIX Byte-Range Lock Enforcement

UNIX provides an API only for byte-range locking. When NFS-mounted data is locked, UNIX uses NLM to transmit the byte-range lock request to the file server. (The filer's NLM implementation supports all of the UNIX byte-range locking semantics, such as region lock merging and sub-region unlocking). SecureShare handles a new NLM byte-range lock request as follows:

- SecureShare first breaks any pre-existing oplock that may be held on the file by a CIFS client. This ensures that SecureShare has all of the information on pre-existing CIFS byte-range locks for the file. (Note: As discussed in Section 4, CIFS clients that hold exclusive or batch oplocks do not propagate lock information back to the server).
- SecureShare next uses Table 2 to check whether the *access-mode* component (*Read* or *Read-Write*) of the lock-mode of the new NLM byte-range lock request conflicts with the *deny-mode* component of any pre-existing file lock-mode. Note that this check is anomalous with respect to the locking hierarchy.
    - If the NLM byte-range lock request is for an exclusive lock (write-lock), it is incompatible with a pre-existing CIFS open having a deny-mode other than *Deny-None.*
    - If the NLM byte-range lock request is for a non-exclusive lock (read-lock), it is incompatible with a pre-existing CIFS open having a deny-mode of *Deny-Read* or *Deny-All*
- SecureShare then checks whether the new NLM byte-range lock request conflicts with any pre-

existing NLM or CIFS byte-range lock that has already been granted on the same file. If a conflict exists, the lock request is rejected.

- If none of the above violations is detected, then the NLM byte-range lock request is granted.

In accordance with NFS/NLM protocol standards, SecureShare treats NLM byte-range locks as advisory with respect to NFS file access functions. However, in accordance with the design goals for its multiprotocol support, SecureShare treats NLM byte-range locks as mandatory with respect to CIFS file access functions.

4. Multiprotocol Oplock Management

CIFS oplocks provide dramatic performance benefits to Windows-only networks. The challenge of a multiprotocol implementation is to provide oplock functionality that compromises neither on data integrity guarantees nor on maximizing accessibility via NFS to oplocked data.

4.1. CIFS Opportunistic Locks

Although the sharing of files and data between multiple Windows clients is fully supported by the CIFS protocol, file sharing between multiple clients is quite rare on most networks. The CIFS protocol leverages the rarity of file sharing in its implementation of the Windows networking performance optimization known as "opportunistic locks" *(oplocks).* An oplock is an exclusive (i.e. *Read-Write/Deny-All*) file-lock that the CIFS client system obtains from the CIFS file server "opportunistically" at application file open time *if* the file being opened is not currently being accessed by any other application. By obtaining an oplock, the client gets a *temporary* exclusive lock on the file being opened, despite the fact that the application did not request an exclusive open. The oplock is *temporary* in the sense that it can be *broken* (i.e. revoked) in case another application tries to gain access to the file. While holding the oplock, the client system takes advantage of the fact that the file is not currently being accessed by any other application. The client's operating system can then — without compromising data integrity — optimize the client/server network traffic needed to satisfy the file accesses by the application. The client holding an oplock minimizes network traffic to the file server by:

1. Performing aggressive read-ahead on open files. Because the client knows that it is the only system accessing the file, it can be certain that the read-ahead data that it obtains is not being changed by other clients back on the server.
2. Aggressively caching write operations to files. There is no need to propagate file changes back to a server synchronously if there are no other clients accessing the file concurrently. Write operations can be delayed and aggregated to optimize I/O performance.
3. Aggressively caching lock requests. The client has no need to inform the server of the various locks an application may have acquired on a file if it can be certain that no other clients are accessing the file concurrently. The locking semantics can be managed entirely on the client side of the connection.

When an application on a CIFS client opens a server-resident file, the client's operating system typically requests an oplock (almost invariably, a type of oplock known as a *batch oplock* that outlives the application's open, as described below) when it transmits a CIFS open request to the file server. If the open was non-exclusive, and the reply to the open did *not* grant an oplock, then the client must synchronously propagate all of the application's interactions with the file back to the file server. In particular, all of the application's write and byte-range lock operations against the file must be synchronously transmitted to the file server. By contrast, if the reply to the open *did* grant an oplock, then the client's operating system can locally-cache write and byte-range lock operations against the file, and can perform aggressive read-ahead on the file.

Oplocks reduce the network traffic transmitted between a CIFS client and the file server, reducing the burden on both network and file server. Note that oplocks are not exposed to Windows applications through the Windows API. They are internal to the CIFS protocol, and are requested automatically from the CIFS file server by the client's operating system at file-open time.

CIFS oplocks are roughly comparable in functionality to the token-based cache synchronization mechanisms of the OSF DCE/DFS distributed file system [10]. On the other hand, the "callback" mechanism of Transarc's AFS is less functional, being equivalent only to DFS "status read" tokens. There is no mechanism for the file server to tell the AFS client to store back to the server any data that the client has modified locally[10].

4.1.1. Oplock Break Protocol in a CIFS-Only Environment

When a second CIFS client attempts to open a file for which there is an outstanding oplock held by an existing CIFS client, the new opener is "held off" while the file server sends the oplock holder an *oplock-break* message. An oplock is actually held by a client operating system, not by an application. Consequently, a client system may choose to hold an oplock for a file even after the application that caused that oplock to be obtained has closed the file and exited. Upon receiving the oplock-break message from the server, the CIFS client operating system can respond in one of two ways:

1. Close the file (after possibly flushing any locally cached write operations on the file back to the server). This would be the client's response in the case of a *batch oplock* when no local opens of the file remain.
2. Flush all outstanding CIFS write and lock operations on the file back to the server, and discard any read-ahead data that it may have obtained for the file. The read-ahead data must be discarded because the second client may subsequently write to the file, invalidating data that the first client originally obtained via read-ahead operations. After it flushes all the cached operations to the server, the client sends the server an *oplock-break-acknowledgement* message, completing the oplock break sequence.

In Case 2, the file remains open. However, the lock-mode of the file-lock representing the open is *down-converted* [8] from lock-mode *Read-Write/Deny-All* to a lock-mode that corresponds to the access- and deny-mode of the original open request. After acknowledging the oplock break, the client must revert to a mode of operation in which it synchronously transmits writes and byte-range lock operations on the file to the server, and in which it refrains from performing file read-ahead.

4.1.2. Exclusive, Batch, and Level II Oplocks

The CIFS protocol defines three types of oplocks: (1) exclusive, (2) batch, and (3) level II.

If a client operating system requests an *exclusive* or *batch* oplock when an application opens a file — and the client is the first and only opener — then the server may grant the client an exclusive or batch oplock. The oplock is represented as a temporary (i.e. *breakable*) file-lock with *Read-Write/Deny-All* lock-mode.

In the case of an *exclusive* oplock, the server then suspends subsequent attempts to open the same file while it breaks the oplock owner's oplock. The oplock break protocol is complete — and the suspended open can proceed (given that its lock-mode is compatible with any file-lock remaining after the oplock break) — once the oplock owner has flushed any outstanding writes to the file server, and has either acknowledged the oplock-break, or has closed the file. Only older CIFS clients appear to use exclusive oplocks. Most CIFS clients use only batch oplocks, a functional superset of exclusive oplocks.

In the case of a *batch* oplock, the server suspends not only subsequent attempts to open the file, but also attempts to remove, rename, or otherwise modify file system metadata associated with the file, while it breaks the oplock owner's oplock. A significant difference between exclusive and batch oplocks is that the latter can outlive file open and close API calls by applications running on the client system. That is, applications on the client system can go through multiple file-open file-close cycles without the client's having to transmit the CIFS open and close requests to the file server. It has been observed that a Windows client will typically request a batch oplock in its CIFS file open request, regardless of the application's requested access- and deny-mode. Acquisition of a batch oplock obviates the client's need to transmit CIFS open and close messages to the server as local applications open and close the file. Unfortunately, some clients will keep a batch oplock *indefinitely* — pending receipt of an oplock break — long after all local openers of the file have exited. This paper terms such a batch oplock "stale."

A level II oplock is — in SecureShare uniform lock-mode terms — a file-lock with lock-mode *Read/Deny-Write*. Holders of level II oplocks on a file — potentially multiple Windows NT clients that may have opened the file for write access, but none of which has been issuing writes — can aggressively cache file read-ahead data. However, they cannot cache file byte-range locks. A client cannot explicitly request a level II oplock at file-open time. However, a Windows NT client opening a file, and requesting an exclusive or batch oplock, can instead be granted a level II oplock. This might occur, for example, if there are multiple concurrent openers, none of which has been issuing any writes. When the first write occurs, all level II oplock openers are *broken to none*; that is, each is sent an oplock break message that revokes its level II oplock, leaving it with no oplock. Alternatively, a former exclusive or batch oplock owner that had originally requested read-only access can be *broken to level II*; that is, it can be sent an oplock break message that revokes its exclusive or batch oplock, leaving it with a level II oplock.

4.2. Extrapolating Oplocks to Multiprotocol

SecureShare extrapolates oplock management to the multiprotocol environment, where UNIX/NFS and/or (PC)NFS clients concurrently access files oplocked by CIFS clients. SecureShare's innovation is to allow non-CIFS accesses to oplocked files — both NFS and NLM requests — to initiate oplock break protocol. The rationale for enabling NLM byte-range lock requests to break oplocks is as follows. Lock-compliant applications issue NLM requests prior to issuing NFS reads or writes. If the NLM request encountered an oplock that it was unable to break, then it would fail, making the file unnecessarily unavailable to the NFS/NLM client. By allowing NFS and NLM accesses to break oplocks, SecureShare ensures that file data remains available to non-CIFS clients while protecting its integrity. A multiprotocol file server that did not enable non-CIFS accesses to break oplocks would have a choice between (1) unnecessarily enforcing a potentially breakable oplock, thereby diminishing file *availability*, or (2) ignoring the oplock, thereby imperiling file data *integrity*. Unnecessary enforcement would cause unreasonable unavailability of the file to NFS/NLM applications in cases where (1) there are no more active openers on the client (i.e. the common case of a stale batch oplock), or (2) there is still an active opener, but its open was non-exclusive. In the case where the oplock is erroneously ignored, the non-CIFS access could lead to data corruption.

4.2.1 Oplock Break due to (PC)NFS

If, after an oplock has been granted to a CIFS client, a (PC)NFS client sends an NLM file-lock request in an attempt to open the file, SecureShare breaks the oplock held by the CIFS client. To complete the oplock break, the first client either closes the file; or else it keeps the file-open, but flushes all its outstanding write and lock operations on the file back to the filer. At this point, the (PC)NFS client's NLM file-lock request can be granted — if it compatible with any file-lock remaining after the oplock break. The (PC)NFS client may then access the file.

4.2.2 Oplock Break due to UNIX/NFS

If, after an oplock has been granted to a CIFS client, a UNIX/NFS client attempts to read or write the open file, SecureShare breaks the oplock held by the CIFS client. If the CIFS client completes the oplock break by closing the file, or else if the file-lock remaining after oplock break completion without a close is compatible with the NFS operation, the NFS request will complete without error. If the NFS request was a read, the read will now access the latest version of the data, as flushed to the filer as a result of the oplock break. If, in the case of an NFS write, the oplock break did not result in the file's being closed, and if the file was opened with *Deny-Write* or *Deny-All*, then the NFS request will fail.

If, after an oplock has been granted to a CIFS client, a UNIX/NFS client attempts to remove or rename the open file, SecureShare breaks the oplock held by the CIFS client. If the CIFS client completes the oplock break by closing the file (i.e. because it was a "stale" batch oplock), then the NFS request is allowed to proceed. If the oplock break did not result in the file's being closed (i.e. because a running Windows application is holding the file open), then the NFS request will fail.

4.2.3 Multiprotocol Oplock Issues

It is worth noting the difference in how SecureShare breaks CIFS oplocks in the contexts of UNIX/NFS data access and (PC)NFS data access to a file. When a UNIX/NFS client attempts to access a file for which an oplock is held by a CIFS client, SecureShare performs the oplock break at the time of the first NFS read or write to the file. However, when a (PC)NFS client attempts to access such a file, SecureShare performs the oplock break at the time the client opens the file. This difference in behavior is due to the fact that, unlike UNIX/NFS clients, (PC)NFS clients request an NLM file-lock at the time a file is opened by a Windows application. Thus SecureShare can detect a (PC)NFS client's intent to access a file at an earlier stage than is possible with a UNIX/NFS client. As the NFS protocol contains no "file-open" operation, SecureShare can only detect a NFS/NLM client's intent to read or write a file at the time that the application actually requests an I/O, or — in lock-compliant applications — requests a byte-range lock.

Without multiprotocol oplock break support in the file server, it would be possible for an NFS client – either UNIX or (PC)NFS based — to corrupt the contents of oplocked files. Such a client could also receive stale, out-of-date data when accessing an oplocked file, even if the Windows application that modified the file exited hours ago (i.e. case of a stale batch oplock). Furthermore, an NFS client could potentially corrupt an oplocked file by NFS-writing into "stale" areas of the file that will later be overwritten when the oplock holder flushes its locally cached file data back to the file. Correct oplock management in a multiprotocol environment is thus seen to be critical to the maintenance of data integrity and cache-coherence.

5. Multiprotocol *Change-Notify*

The CIFS protocol contains a feature, *change-notify*, that enables an application running on a CIFS client to request that the server notify the client whenever a change occurs in one or more directories that the client wishes to monitor. The application can monitor either a single directory or the entire file system subtree under a directory. The application supplies a parameter specifying the kinds of changes to be monitored. Examples of directory-relative events that can be monitored are file/directory creation, deletion, rename/move, attribute change, modification time change, etc. The notification takes the form of a server-to-client message containing potentially multiple entries, each of which specifies the name of a changed file or sub-directory within the monitored directory, together with the type of change. When one client changes a directory that is being monitored by other clients, the monitoring clients are notified of the change so that they can take appropriate action, such as updating a GUI display.

SecureShare extrapolates *change-notify* to the cross-platform, multiprotocol environment. With multiprotocol *change-notify*, modification to a monitored directory by *either CIFS or NFS* triggers the asynchronous transmission of change notification messages to the *change-notify* clients.

SecureShare's cross-platform *change-notify* facility enables independent software vendors to utilize Network Appliance filers in developing cooperating, cross-platform applications that communicate through the filer's file system. UNIX applications can process files, and then "hand them off" to Windows NT applications for further processing. Cross-platform *change-notify* allows these applications to accomplish the "hand-off" simply by copying the files into monitored directories.

The *change-notify* mechanism is implemented using a SecureShare-proprietary type of file-lock, a *change monitoring file-lock*. When a Windows NT application uses the *change-notify* API, CIFS sends a directory file-open, followed by a CIFS *change-notify* request. These CIFS requests cause SecureShare to generate a change monitoring file-lock on the monitored directory. The change monitoring file-lock allows Data ONTAP to efficiently test for the potential need to send change notifications whenever the kernel executes a CIFS or NFS request that makes changes within a directory.

6. Summary

SecureShare's approach to multiprotocol network storage fully effectively addresses the data integrity, cache-coherence, and file-sharing issues inherent in a cross-platform file-sharing environment. SecureShare protects shared data from the various challenges to data integrity and cache-coherence that arise when the same files and directories are being read and written concurrently by CIFS and NFS/NLM clients. It does this by reconciling the different and incompatible semantics utilized by CIFS and NFS/NLM clients for file locking, file-open, and file sharing. By allowing NFS and NLM requests to break CIFS oplocks, it ensures that file data remains available to NFS clients while simultaneously protecting the integrity and cache-coherence of that data. SecureShare accomplishes these goals by means of a uniform approach to managing all the locking paradigms in the UNIX and Windows environments.

7. Acknowledgement

I am deeply indebted to Pei Cao, Assistant Professor of Computer Science, University of Wisconsin, Madison, who served as my conference paper "shepherd." She reorganized this paper, greatly improving the presentation of the material.

8. References

[1] Brown, K., A. Borr: "SecureShare: Guaranteed Multiprotocol File Locking," Technical Report 3024, Network Appliance, Inc., November 1997.

[2] Hitz, D.: "An NFS File Server Appliance," Technical Report 3001, Network Appliance, Inc., 1995.

[3] Sun Microsystems, Inc: "NFS: Network File System Protocol Specification," RFC-1094, DDN Network Information Center, SRI International, March 1989.

[4] Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, B. Lyon: "Design and Implementation of the Sun Network Filesystem," USENIX Conference Proceedings, USENIX Association. Summer 1985.

[5] X/Open Company, Ltd.: "X/Open CAE Specification: Protocols for X/Open Interworking: XNFS," X/Open Company, Ltd., 1991.

[6] Internet Engineering Task Force Network Working Group: "A Common Internet File System (CIFS/1.0) Protocol", ftp://ds.internic.net/internet-drafts/draft-leach-cifs-v1-spec-01.txt, December 1997.

[7] X/Open Company, Ltd.: "X/Open CAE Specification: Protocols for X/Open Interworking: (PC)NFS," X/Open Company, Ltd., 1991.

[8] Gray, J: "Notes on Data Base Operating Systems," IBM Research Report RJ2188, IBM San Jose Research Laboratory, February 1978.

[9] The Samba Team: "Samba: A LanManager like SMB fileserver for UNIX," home page: http://samba.anu.edu.au/samba.

[10] Kazar, M., et. al.: "DEcorum File System Architectural Overview," USENIX Conference Proceedings, USENIX Association, June 1990.

		Existing file lock-mode										
		NULL	A: R D: DN	A: R D: DR	A: R D: DW	A: W D: DN	A: W D: DR	A: W D: DW	A: RW D: DN	A: RW D: DR	A: RW D: DW	A: Any D: DA
New mode being requested	A: R D: DN	✓	✓	X	✓	✓	X	✓	✓	X	✓	X
	A: R D: DR	✓	X	X	X	✓	X	✓	X	X	X	X
	A: R D: DW	✓	✓	X	✓	X	X	X	X	X	X	X
	A: W D: DN	✓	✓	✓	X	✓	✓	X	✓	✓	X	X
	A: W D: DR	✓	X	X	X	✓	✓	X	X	X	X	X
	A: W D: DW	✓	✓	✓	X	X	X	X	X	X	X	X
	A: RW D: DN	✓	✓	X	X	✓	X	X	✓	X	X	X
	A: RW D: DR	✓	X	X	X	✓	X	X	X	X	X	X
	A: RW D: DW	✓	✓	X	X	X	X	X	X	X	X	X
	A: Any D: DA	✓	X	X	X	X	X	X	X	X	X	X

Table 1

Lock Compatibility Matrix

Used to check whether a requested lock-mode is compatible with a pre-existing file-lock.

A = Access-Mode (R = Read, W = Write, RW = Read-Write, Any = any one of R or W or RW)

D = Deny-Mode (DN = Deny-None, DR = Deny-Read, DW = Deny-Write, DA = Deny-All)

✓ = New open request will be **granted.** **X** = New open request will be **denied.**

	Existing file lock-mode										
	None	A: R D: DN	A: R D: DR	A: R D: DW	A: W D: DN	A: W D: DR	A: W D: DW	A: RW D: DN	A: RW D: DR	A: RW D: DW	A: Any D: DA
NLM Write Lock	✓	✓	X	X	✓	X	X	✓	X	X	X
NLM Read Lock	✓	✓	X	✓	✓	X	✓	✓	X	✓	X

Table 2

Compatibility of NLM Byte-Range Locks with CIFS File-Locks

✓ = New request will be **granted.** **X** = New request will be **denied.**

Harnessing User-Level Networking Architectures for Distributed Object Computing over High-Speed Networks

Rajesh S. Madukkarumukumana
Server Architecture Lab
Intel Corporation
5200 N.E. Elam Young Pkwy
Hillsboro, OR 97124
rajesh@co.intel.com

Calton Pu
Department of Computer Science & Engineering
Oregon Graduate Institute (OGI) of Science and Technology
Portland, OR 97291
calton@cse.ogi.edu

Hemal V. Shah
Server Architecture Lab
Intel Corporation
5200 N.E. Elam Young Pkwy
Hillsboro, OR 97124
hvshah@co.intel.com

Abstract

In a distributed object system such as Distributed Component Object Model (DCOM) [5, 7], legacy transport protocols used for communication limit the performance over high-speed networks. By making use of a low-latency, high-bandwidth, and low overhead user-level networking architecture such as Virtual Interface (VI) Architecture [8, 18], this performance bottleneck can be significantly reduced. Since user-level networking architectures provide low-level primitives, the challenge lies in integrating them into high-level applications. This requires a systematic approach. In this paper, a methodology to utilize VI Architecture to improve the performance of DCOM using custom object marshaling is developed. Initial experimental results demonstrate that the latencies of small messages in distributed object computing can be significantly reduced by this methodology.

Keywords: Virtual Interface (VI) Architecture, User-level Networking Architecture, Distributed Component Object Model (DCOM), Distributed Object Computing, Custom Object Marshaling.

1. Introduction

Component based software offers modularity, reduces applications' integration and maintenance costs, and improves deployment flexibility. Distributed object frameworks like Distributed Component Object Model (DCOM) [7], Common Object Request Broker Architecture (CORBA) [16], and Java Remote Method Invocation (RMI) [13] facilitate building distributed applications from simple components. Distributed object frameworks use remote procedure call (RPC) mechanism to perform remote object activations and remote method invocations. The overheads associated with underlying legacy transport protocols (e. g. UDP, TCP) used in RPC mechanisms introduce considerable latency over high-speed networks such as System Area Networks (SANs).

User-level networking architectures, such as the Virtual Interface (VI) Architecture [8, 18], U-Net [10], and SHRIMP Virtual Memory Mapped Communication (VMMC) [2] that are designed to achieve low-latency and high-bandwidth in a SAN environment, offer an attractive solution for reducing communication software overheads. Building high-level applications, using low-level primitives offered by user-level networking architectures, is complex. This paper focuses on the challenge in integrating user-level networking architectures into distributed object frameworks. In this research, DCOM is the target distributed object model and VI Architecture is used as the user-level networking architecture. This paper provides the following two contributions in harnessing user-level networking architectures for distributed object computing over high-speed networks:

- A specialization methodology to replace legacy RPC transports in DCOM with VI-based transport for SAN environments,
- Latency analysis of standard and VI-enabled DCOM remoting architecture.

Integration of VI into DCOM remoting architecture is achieved by custom object marshaling mechanism. This involves specialization of object implementation and generation of custom proxy/stub code along with marshaling routines. Initial experimental results provide evidence of the performance improvement.

The organization of the rest of the paper is as follows. Brief overviews of VI Architecture and DCOM are provided in Section 2 and Section 3 respectively. In Section 4, a mechanism to integrate VI Architecture into DCOM to reduce remote method invocation latencies is discussed. Experimental results are provided in Section 5. Section 6 briefly summarizes some related work. Finally, future work is discussed and conclusion is drawn in Section 7.

2. Virtual Interface Architecture

VI Architecture is a user-level networking architecture designed to achieve low latency, high bandwidth communication between processes running on nodes connected by a high-speed network within a computing cluster. To a user process, the VI Architecture provides direct access to the network interface in a fully protected fashion. The VI Architecture avoids intermediate copies of the data and bypasses operating system to achieve low latency, high bandwidth data transfer. The VI Architecture Specification 1.0 [18] was jointly authored by Intel Corporation, Microsoft Corporation, and Compaq Computer Corporation.

The VI Architecture uses a VI construct to present an illusion to each process that it owns the interface to the network. A VI is owned and maintained by a single process. Each VI consists of two work queues: one send queue and one receive queue. On each work queue, Descriptors are used to describe work to be done by the network interface. A linked-list of variable length Descriptors forms each queue. Ordering and data consistency rules are only maintained within one VI but not between different VIs. VI Architecture also provides a completion queue construct that is used to link completion notifications from multiple work queues to a single queue.

Memory protection for all VI operations is provided by protection tag (a unique identifier) mechanism. Protection tags are associated with VIs and memory regions. The memory regions used by Descriptors and data buffers are registered prior to data transfer operations. Memory registration gives VI NIC a method to translate virtual addresses to physical addresses. The user receives an opaque memory handle as a result of memory registration. This allows user to refer to a memory region using a memory handle/virtual address pair without worrying about crossing page boundaries and keeping track of the mappings of virtual addresses to tags.

The VI Architecture defines two types of data transfer operations: 1) traditional send/receive operations, and 2) Remote-DMA (RDMA) read/write operations. A user process posts Descriptors on work queues and uses either polling or blocking mechanism to synchronize with the completed operations. The two Descriptor processing models supported by VI Architecture are the work queue model and the completion queue model. In the work queue model, the VI consumer polls or waits for completions on a particular work queue. The VI consumer polls or waits for completions on a set of work queues in the completion queue model. The processing of Descriptors posted on a VI is performed in FIFO order but there is no implicit relationship between the processing of Descriptors posted on different VIs.

For more details on VI Architecture, the interested reader is referred to [8, 18]. Figure 2 compares the one-way latency of UDP with one-way latency of software emulated VI (in host driver) over a 100 Mbps Ethernet. The latencies were measured using ping-pong tests and were averaged over 1000 runs. Figure 2 illustrates that even the latency of VI emulated in host driver is significantly less than the latency of UDP. With VI functionality implemented in NIC hardware, the latency can be significantly reduced further.

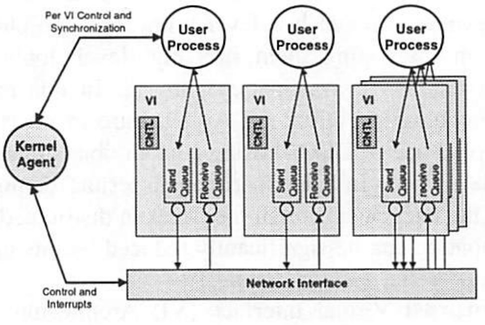


Figure 1: VI Queues

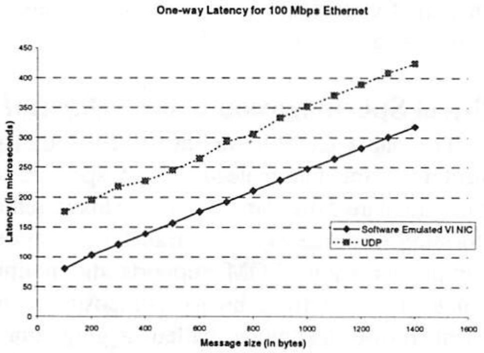


Figure 2: Emulated-VI vs. UDP Latency

3. Distributed Component Object Model (DCOM)

The Component Object Model (COM) [5] is an architecture and a supporting infrastructure for creating, using and evolving component software and building applications using these components. COM provides a binary standard to which components and its clients must adhere in order to ensure dynamic interoperability. Distributed Component Object Model (DCOM) [7] is an extension to COM for networked environments to support distributed computing. The overall DCOM architecture consists of the COM programming interface, the interface remoting infrastructure, and the wire protocol. COM allows clients to communicate with an object solely through the use of *vtable*-based interface instances. This provides a single programming model for accessing in-process, local and remote components. The interface remoting infrastructure in COM facilitates this location transparency. The DCOM wire protocol describes the content and the format of what is actually transmitted across the network when components reside on remote machines.

3.1 DCOM Architecture

The marshaling architecture in DCOM performs encoding and decoding of method call/return parameters into a standard data representation (marshaling and unmarshaling) that can be sent across the network. DCOM remoting architecture is abstracted as an Object RPC (ORPC) layer built on top of DCE RPC infrastructure. DCE RPC defines the standard data representation (NDR) for all relevant data types.

Interface pointers in COM are either returned from object activations or passed as parameters in method calls. COM has a special data type not present in DCE RPC to handle interface pointers in a uniform way. Marshaling and unmarshaling of COM interface pointers entails creation of a stub object in the server process and a proxy object in the client process respectively. Proxy and stub are capable of handling remote method invocations to the marshaled interface.

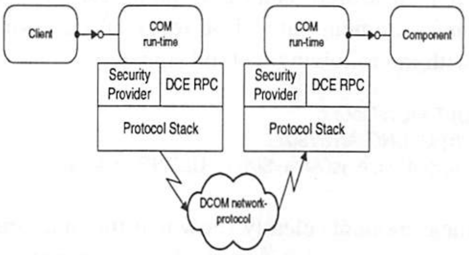


Figure 3: DCOM Architecture

The COM Specification defines various types of interface pointer marshaling, namely, standard marshaling, handler marshaling, and custom marshaling. Standard marshaling in COM provides the glue to the underlying RPC infrastructure and allows the component and the client to be completely ignorant of the marshaling and remoting architecture. Compiling the component's Interface Description Language (IDL) file with the MIDL compiler generates the proxy and stub code for standard marshaling. Handler marshaling extends the COM marshaling architecture by allowing the component to plug-in smart handlers that can intercept client's method calls and choose to satisfy them or forward them to the standard proxy. The design of an interface that focuses only on its function can lead to design decisions that conflict with efficient implementation across a network. In cases like these, COM allows object implementers to extend or even override standard marshaling of an interface pointer by the use of custom marshaling. Custom marshaling maintains complete client transparency. This architectural extensibility makes it possible to address network performance issues without disrupting the established design. For more details on COM and DCOM architectures, the interested reader is referred to [5, 7].

Custom marshaling allows the object to dynamically choose how its interface pointers are marshaled. Custom object marshaling is useful in many techniques including:

- replacing COM ORPC with other transports,
- marshaling static objects by value,
- adding fault-tolerance and high-availability properties to objects,
- performing replication transparently to the client and the component.

Wang et. al. briefly described some of these techniques in [19]. RPC infrastructure used in COM standard marshaling can work over a variety of legacy transport protocols like UDP, TCP, etc. Due to the inherent scalability offered by UDP, it is the default (and most widely used) DCOM protocol. Figure 4 shows the one-way latency (averaged over 1000 runs) of COM, RPC and UDP measured using ping-pong tests. DCOM and RPC measurements used bi-directional conformant arrays with the following method signature:

```
HRESULT MoveData (
     [in] ULONG ArraySize,
     [in, out, size_is(ArraySize)]  ULONG *pArray );
```

The measurements clearly show that for small messages (common case in distributed object computing frameworks), latency incurred in RPC and DCOM is dominated by UDP latency.

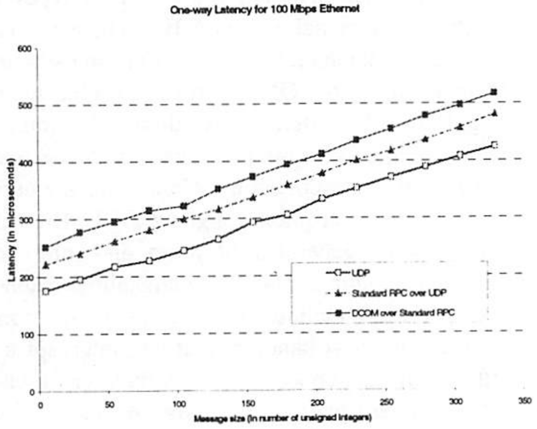


Figure 4: UDP, RPC, and DCOM Latency

4. DCOM Remote Method Invocation over VI Architecture Transport

COM marshaling architecture is extensible through its implementation of proxies and stubs as in-process COM servers. A COM object implementation advertises its ability to perform custom marshaling by exporting a standard COM interface called *IMarshal*. An object that does not export the *IMarshal* interface gets the standard proxy and stub by default. As part of marshaling an interface pointer, COM allows objects to perform any arbitrary action (like creating a custom stub) and to provide any block of data representing the custom object reference. The object can also specify the class identity (CLSID) of the custom proxy that can unmarshal the custom object reference on the client side.

Upon receiving the marshaled data, COM runtime instantiates the specified custom proxy in the client process. The custom proxy uses the marshaled object reference data to setup a connection to the stub and exposes the same *vtable* representation of the remoted interface to the client. Figure 5 illustrates a custom marshaling architecture that uses the high performance user-level VI transport for inter-process communications. The architectural details are described next.

4.1 Object Specialization using *IMarshal*

To enable COM remoting over VI transport, the object implementation needs to be specialized to expose the standard *IMarshal* interface. This is achieved by performing a source-to-source transformation of the object implementation. COM supports the notion of composing an object from binary composites using a component re-use technique called aggregation [5]. COM aggregation is useful in specialization as it allows composing objects dynamically. To minimize the source transformation needed to expose the *IMarshal* interface, the specialized object aggregates *IMarshal* from the inner custom stub. The specialization process is automated by a *"Custom Marshaling Wizard"* integrated into the Microsoft Developer Studio environment as a *"DevStudio Add-In"* component.

In the current prototype, COM automation interfaces, non-C++ object implementations, and VI Remote DMA (RDMA) operations are not supported. In order to support co-existence of standard and custom remote proxies and to preserve object identity, the available context information needs to be extended by using either 'channel hooks' [9] or custom class factories. Security features are not present in the current custom marshaler, but can be provided using standard Windows NT challenge/reply authentication.

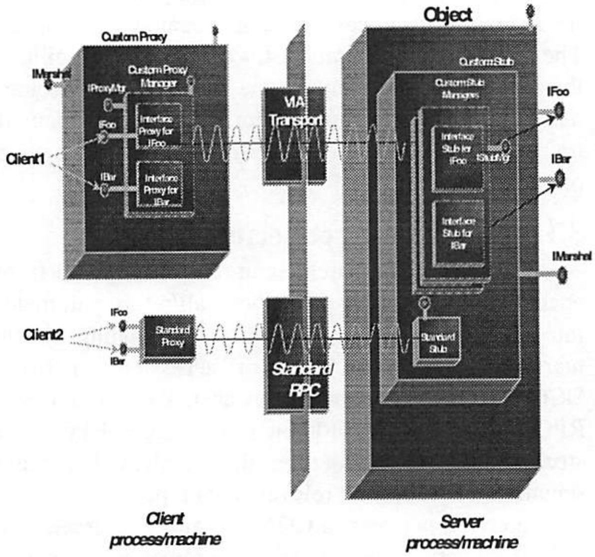


Figure 5: Custom Object Marshaling

4.2 Anatomy of Custom Stub/Proxy

A custom stub (proxy) is itself a COM object. A custom stub consists of a stub manager and interface stubs for each of the component's marshaled interfaces. Each custom stub manager represents an endpoint connection from a specific remote client process to the marshaled object. A custom stub manager manages endpoint creation and destruction, data transfers, and object lifetime. It also dispatches method requests to interface stubs. The custom stub uses the context and marshal flags passed as *IMarshal* method parameters to delegate unsupported contexts (e.g. table marshaled and local objects) to the standard marshaler. Each interface stub unmarshals method parameters from receive buffers, dispatches actual object methods, marshals return parameters into the reply buffers, and returns to the stub manager. The stub manager sends the reply buffer to the client. On the client side, each custom proxy is a peer to the corresponding custom stub and consists of proxy manager and interface proxies.

Marshaling method parameters into the standard data representation (NDR) provides heterogeneity and allows application programmers to use any user-defined data structures as method parameters. NDR marshaling (*Pickling*) used by the custom proxies and stubs avoids intermediate buffer copies by marshaling method parameters directly into registered transmission buffers. Procedural encoding is used to avoid buffer packing complexities, and incremental encoding is used to meet dynamic memory requirements. The NDR routines are generated from a transformed IDL file derived from the original application IDL file using the following rules:

1. Each method is split into a request method and a reply method. The request method contains all the method parameters passed from the client to the object. The reply method contains parameters returned from the object to the client including the HRESULT. Any parameter that is used in attributes of return parameters is also included in the reply method.
2. Since request and reply method parameters are marshaled at one end and unmarshaled at the other end, each parameter is declared bi-directional and a level of indirection is added to it. The added indirection is propagated in parameter attributes as well.
3. Marshaling of interface pointers in method parameters are handled separately as Windows™ NT encoding services do not currently support them.
4. Custom proxies and stubs support a set of special interfaces to allow marshaling of interface pointers that are references to custom proxies.

The following example shows an IDL transformation using some of the above rules.

```
Original Method's Signature:
    HRESULT MoveData(
        [in] ULONG ArraySize,
        [in, out, size_is(ArraySize)]
        ULONG *pArray );

Transformed Methods' Signatures:
    void MoveData_Request(
        [in, out]  ULONG *ArraySize,
        [in, out, unique, size_is(, *ArraySize)]
        ULONG **pArray );
    void MoveData_Reply(
        [in, out] HRESULT *ReturnCode,
        [in, out]    ULONG *ArraySize,
        [in, out, unique, size_is(, *ArraySize)]
        ULONG **pArray );

Generated NDR Routines:
    void MoveData_Request(
        handle_t IDL_handle,
        ULONG *ArraySize,
        ULONG ** pArray );
    void MoveData_Reply(
        handle_t IDL_handle,
        HRESULT *ReturnCode ,
        ULONG *ArraySize,
        ULONG **pArray );
```

By running the MIDL compiler over the transformed IDL file along with a supporting application configuration file (ACF), the NDR encoding routines are generated. The custom proxy (stub) dynamic link library (DLL) is created from the generated NDR routines, interface proxy (stub) templates, and the static proxy (stub) manager code. The whole process of custom proxy and stub generation can be automated by integrating it into the *"Custom Marshaling Wizard"*. Figure 10 provides a snapshot of *"Custom Marshaling Wizard"*.

5. Experimental Results

In order to demonstrate DCOM performance improvements achieved by integrating user-level VI transports, a set of experiments was carried out. In the experiments, a pair of server systems, with dual 200 MHZ Pentium ® Pro processors (with 256K L2 cache), Intel 82440FX PCI chipset, and 64 MB memory, was used as a pair of host nodes. Intel Pro100B Ethernet (100 Mbps) NIC with VI functionality emulated in software (host driver), Myricom's Myrinet [3] NIC (1.28 Gbps) with VI functionality emulated on NIC firmware, and GigaNet's cLAN™ GNN1000 interconnect (1.25 Gbps full duplex) [11] with VI functionality implemented on NIC hardware were used as VI NIC prototypes. The software environment used for all the experiments included Windows™ NT 4.0 with service pack 3 and Microsoft Visual C++ 5.0.

VI and UDP latency tests measure the time to copy the contents from an application's data buffer to another application's data buffer across an interconnect using a round-trip (ping-pong) test. DCOM and RPC latency measurements used bi-directional conformant arrays as method parameters with the method signature described in Section 4.2. VI architecture provides both polling and blocking models for synchronization. In the polling model, the user thread directly polls on the status of descriptors posted on VI work queues, thereby avoiding interrupt generation and processing overheads at the cost of increased CPU utilization. Reducing interrupts has a significant impact on the capacity of the system in addition to reducing the per-packet send/receive latencies. For GigaNet VI NICs, the experiments were carried out for both polling and blocking models. In the experiments involving Ethernet and Myrinet, only polling model was used.

In all the experiments, COM servers and clients used were free-threaded and COM security features were disabled. In case of custom stubs and proxies, method parameters and other information (including the NDR header) are marshaled and unmarshaled directly into and out of registered send/receive communication buffers to avoid intermediate data copies.

The VI Architecture is designed to enable applications to communicate over a SAN that provides high bandwidth, low latency communication with low error rates. At the NIC level, the VI Architecture provides three levels of reliability: Unreliable Delivery, Reliable Delivery, and Reliable Reception. Only VIs with the same reliablility level can be connected. In the experiments, the level of reliability used in each VI was Reliable Delivery. According to [18], this level of reliability guarantees that all data submitted to a reliable delivery VI will arrive at its destination exactly once, intact, and in the order submitted, in the absence of errors. For this level of service, transport errors are considered catastrophic and should be extremely rare. Due to this level of service used along with low error rates on high-speed networks, error recovery in form of application specified timeouts was incorporated in custom marshaled proxies and stubs.

Figures 6 and 7 compare one-way COM remote method invocation latencies (averaged over 1000 runs) between the standard and the specialized components across Ethernet and Myrinet interconnects respectively. Since the VI functionality was emulated either in host driver or in NIC firmware, the performance measurements are conservative.

From Figures 6 and 7, it is clear that VI-based communication in DCOM substantially reduces the performance bottleneck due to the use of legacy protocol stack. In the case of Ethernet, one-way latency was reduced by more than 150 microseconds (30% - 60 %) by using VI-based communication in DCOM. Due to unavailability of the standard Windows™ NT NDIS driver on Myrinet, UDP-based measurements were not obtained. Interestingly, Figure 7 indicates that on Myrinet the performance of DCOM over VI is better than that of COM over local RPC for small messages (≤ 2 KB). The VI-emulation in Ethernet driver is useful for proof-of-concept validation, but it achieves only limited performance. Even though the VI-emulation on Myrinet NIC performs better, the slow (33MHZ) on-board controller (MCP) limits the overall gain.

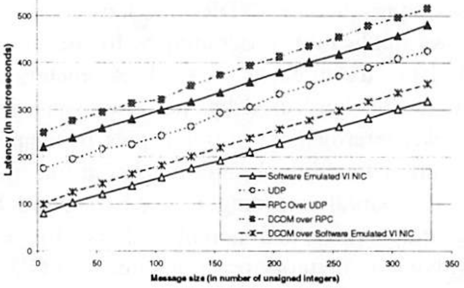


Figure 6: DCOM over VI on Ethernet (Polling Model)

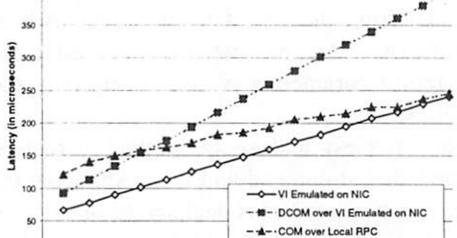


Figure 7: DCOM over VI on Myrinet (Polling Model)

Figures 8 and 9 show the results of similar experiments over the GigaNet cLAN™ GNN1000 native VI NICs using blocking and polling synchronization models respectively. The results obtained over GigaNet VI NICs confirm that availability of core VI functionality in special purpose hardware on network adapters can significantly improve communication performance. Figures 8 and 9 also demonstrate that the specialization methodology developed in this paper can deliver the raw performance offered by these high-speed interconnects to higher level COM applications.

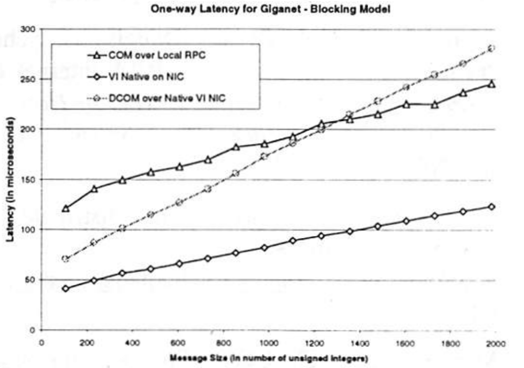


Figure 8: DCOM over VI on GigaNet (Blocking Model)

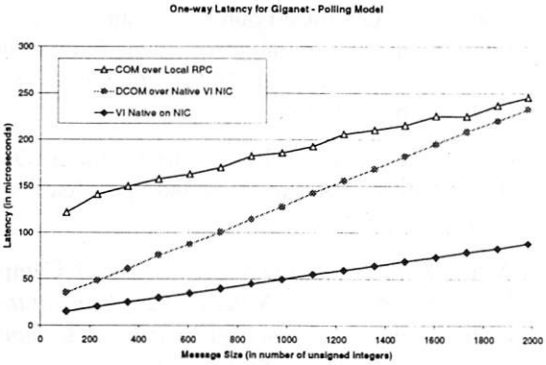


Figure 9: DCOM over VI on GigaNet (Polling Model)

The NDR processing continues to be a significant part of the remaining COM remote method invocation overhead (this was verified by using Intel VTune™ performance monitoring tool) and is a good candidate for further optimization using other specialization techniques like partial evaluation as proposed by Muller et. al. [15]. Figures 8 and 9 also indicate that the DCOM performance over GigaNet VI NICs is better than the performance of COM over local RPC for small messages (≤ 5 KB for blocking model and ≤ 8 KB for polling model).

For both GigaNet and Myrinet, DCOM over VI outperformed COM over local RPC for small messages. In case of COM over local RPC, communication between proxy and stub involves overheads of context switching and synchronization. On the other hand, since proxy and stub reside on different nodes in DCOM, the cost of context switching between proxy and stub is eliminated. The advantage of VI Architecture here is that the VI NIC performs the tasks of multiplexing, demultiplexing, and data transfer scheduling normally being performed by an OS kernel and device driver in legacy transports. Thus, OS overheads are significantly reduced in case of DCOM over VI. This suggests that for small messages, with high-speed interconnects and low overhead user-level networking architecture like VI Architecture, the cost of moving data between two processes residing on different nodes can be less than the cost of moving data between two local processes residing on the same node.

6. Related Work

Application level optimizations such as application level framing and integrated layer processing are used by Schmidt et. al. [12] to reduce CORBA latency. COMERA [19] proposes an extensible COM remoting architecture for transparent fault tolerance, migration, and replication properties. Quarterware kit [17] enables building middleware implementations that can be customized for performance and additional features. Muller et. al. [15] showed how specialization techniques like partial evaluation can be applied to improve RPC performance. All of the above approaches use application level optimizations but do not address utilizing user-level networking architectures to improve performance. Damianakis et. al. [6] pointed out that performance of higher level programming models such as stream sockets and remote procedure calls can be improved by using a user-level networking architecture. This paper adds on to their findings by improving object middleware performance using VI Architecture.

7. Future Work and Conclusion

Object specialization through custom object marshaling requires modifications to the object implementation, even though this can be fully automated. By adding a new protocol sequence for VI Architecture in addition to current suite of RPC protocols (ncadg_ip_udp, ncacn_ip_tcp, etc.) complete client and object transparency can be achieved. We are currently investigating this approach.

In this paper, custom marshaling based object specialization methodology was developed to integrate high-performance user-level networking architectures into distributed component object model (DCOM). The experimental results confirm that high-performance provided by VI Architecture can be delivered to high level COM applications using this specialization methodology. Standard high volume servers (SHVs), commodity high-speed interconnects, and standard based user-level networking architectures like VI Architecture can open new horizons to off-the-shelf distributed applications by providing high performance at low cost.

Acknowledgements

We would like to acknowledge David Fair, Roy Larsen, Wire Moore, Greg Regnier, and Mitch Shults of Intel Corporation for their many useful suggestions and reviews.

References

1. Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber, Network Objects, DEC SRC Research Report 115.

2. M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg, " A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer", *Proc. of the 21st Annual Symposium on Computer Architecture*, 1994, pp. 142-153.

3. N. J. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and Wen-King Su, "Myrinet: A Gigabit-per-Second Local-Area Network", *IEEE MICRO,* Vol. 15, No. 1, February 1995, pp. 29-36, http://www.myri.com/research/publications/Hot.ps

4. P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. Shih, C.-Y. Wang, and Y.-M. Wang, "DCOM and CORBA Side by Side, Step by Step, and Layer by Layer", *C++ Report*, January 1998.

5. Component Object Model Specification, Microsoft Corporation, 1995. http://www.microsoft.com/com

6. Stefanos Damianakis, Angelos Bilas, Cezary Dubnicki, and Edward W. Felten, "Client Server Computing on SHRIMP", *IEEE MICRO*, January/February 1997, Vol. 17, No. 1.

7. DCOM Architecture, Microsoft Corporation, 1997.

8. Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, Ellen Deleganes, David Fair, Chris Dodd, and Justin Rattner, "The Virtual Interface Architecture: A Protected, Zero Copy, User-level Interface to Networks", *IEEE MICRO,* March/April 1998, Vol. 18, No. 2, pp. 66-76.

9. Guy Eddon and Henry Eddon, "Understanding the DCOM Wire Protocol by Analyzing Network Data Packets", *Microsoft Systems Journal*, March 1998, pp. 45-63.

10. Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing", *Proc. of the 15th ACM Symposium on Operating System Principles*, 1995, pp. 40-53.

11. GigaNet Incorporated, GigaNet cLAN Product Family, http://www.giga-net.com/products

12. Aniruddha S. Gokhale and Douglas C. Schmidt, "Principles for Optimizing CORBA Internet Inter-ORB Protocol Performance", *Proc. of Hawaii International Conference on System Sciences (HICSS)*, January 1998.

13. Java Remote Method Invocation – Distributed Computing for JAVA, http://java.sun.com/marketing/collateral/javarmi.html

14. Mary Kirtland, "The COM+ Programming Model Makes it Easy to Write Components in Any Language", *Microsoft System Journal*, December 1997.

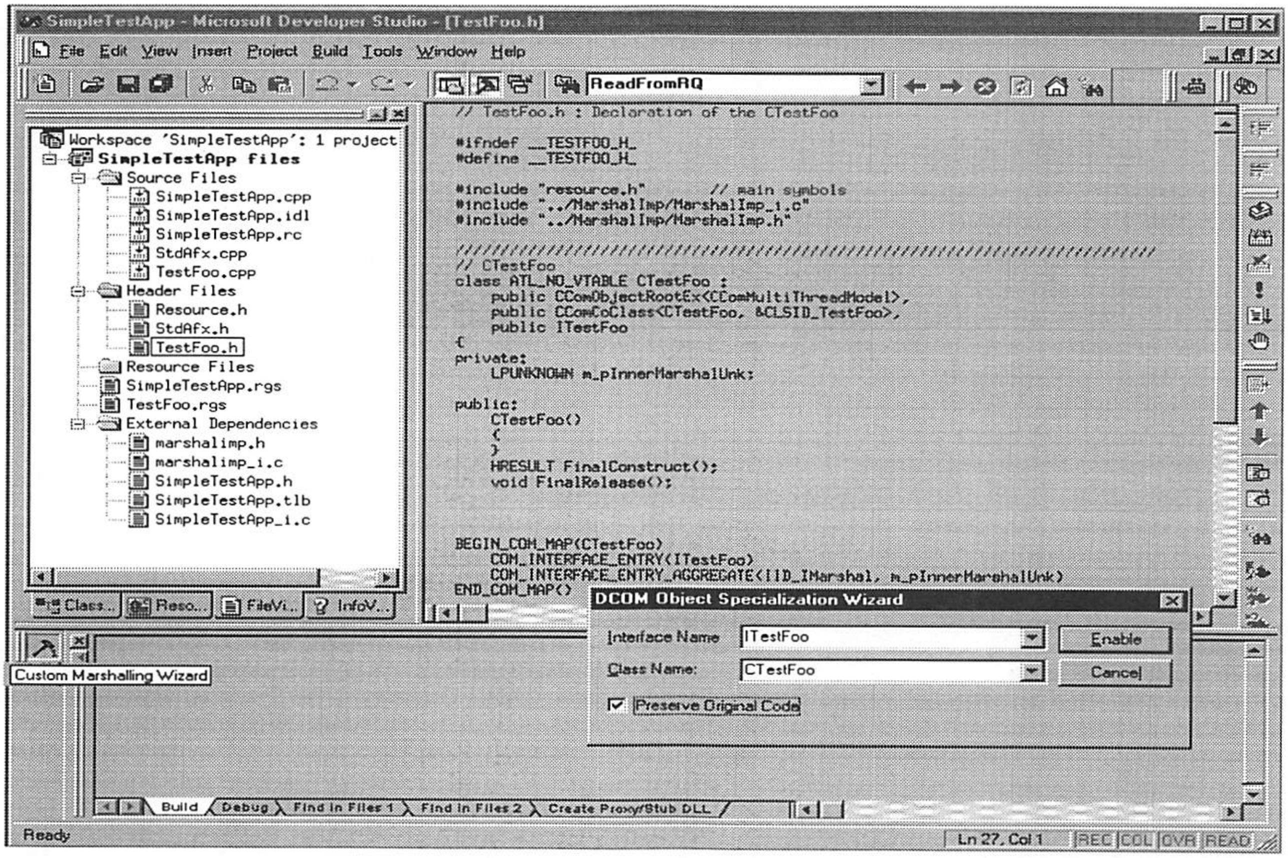
15. Gilles Muller, Renaud Marlet, Eugen-Nicolae Volanschi, Charles Consel, Calton Pu and Ashvin Goel, "Fast, Optimized Sun RPC Using Automatic Program Specialization", *Proc. of the International Conference on Distributed Computing Systems (ICDCS-18)*, May 1998.

16. Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.0 edition, July 1995.

17. Ashish Singhai, Aamod Sane, and Roy H. Campbell, "Quarterware for Middleware", *Proc. of the International Conference on Distributed Computing Systems (ICDCS-18)*, May 1998.

18. Virtual Interface Architecture Specification, Version 1.0, December 1997. http://www.viarch.org/

19. Yi-Min Wang and Woei-Jyh Lee, "COMERA: COM Extensible Remoting Architecture", *Proc. of 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, April 1998. http://www.research.att.com/~ymwang/papers/HTML/COMERA/S.html



```
// TestFoo.h : Declaration of the CTestFoo

#ifndef __TESTFOO_H_
#define __TESTFOO_H_

#include "resource.h"       // main symbols
#include "../MarshalImp/MarshalImp_i.c"
#include "../MarshalImp/MarshalImp.h"

/////////////////////////////////////////////////////////////////////////////
// CTestFoo
class ATL_NO_VTABLE CTestFoo :
    public CComObjectRootEx<CComMultiThreadModel>,
    public CComCoClass<CTestFoo, &CLSID_TestFoo>,
    public ITestFoo
{
private:
    LPUNKNOWN m_pInnerMarshalUnk;

public:
    CTestFoo()
    {
    }
    HRESULT FinalConstruct();
    void FinalRelease();


BEGIN_COM_MAP(CTestFoo)
    COM_INTERFACE_ENTRY(ITestFoo)
    COM_INTERFACE_ENTRY_AGGREGATE(IID_IMarshal, m_pInnerMarshalUnk)
END_COM_MAP()
```

DCOM Object Specialization Wizard

Interface Name: ITestFoo

Class Name: CTestFoo

☑ Preserve Original Code

Enable Cancel

Figure 10: A Snapshot of "Custom Marshaling Wizard"

Implementing IPv6 for Windows NT

Richard P. Draves,[1] Allison Mankin,[2] Brian D. Zill[1]

[1]*Microsoft Research*
One Microsoft Way
Redmond, WA 98052
richdr@microsoft.com

[2]*USC/ISI*
4350 North Fairfax Drive
Arlington, VA 22203
mankin@isi.edu

Abstract

We have created a publicly-available implementation of IPv6 for Windows NT. Because we have made our source code available, we hope that our implementation can serve as a base for networking research and supply sample code for other implementations. In this paper we describe our solutions for several problems that any network protocol implementation for Windows NT will encounter. Based on our experience, we also comment on the utility of access to the source code for the Windows NT product.

1. Introduction

This paper reports our experiences developing a network protocol stack for Windows NT. We have created a prototype implementation, known as MSR IPv6, of IPv6 for Windows NT. We have released the implementation and its source code publicly, for testing, research, and educational purposes [10]. Our implementation should prove useful to people experimenting with IPv6 and to people wishing to use Windows NT as a platform for networking research or education.

IPv6 [3] is the next version of the Internet Protocol; it is an active effort within the Internet Engineering Task Force (IETF). IPv6 primarily solves scaling problems with the current version of the Internet Protocol (IPv4), but it also introduces many other major and minor architectural improvements. Most notably, IPv6 addresses have 128 bits (16 bytes) [7]. We will introduce other salient aspects of IPv6 throughout the paper, as they are relevant. Numerous vendors have pre-release IPv6 implementations and there are several free, publicly available implementations for BSD and Linux variants [9]. Our implementation is the first free, publicly available implementation for Windows NT.

We started this project at Microsoft Research in late 1996 primarily as a learning experience: we wanted to learn more about the Internet Protocol and this step in its evolution, and we wanted to learn more about Windows NT internals. We also hoped that our efforts might help bootstrap a Microsoft product implementation of IPv6. (We can not say anything further about Microsoft's product plans or schedules.) As we made progress, we realized that a public release, including source code, would be valuable for the community. USC/ISI East joined the project in December 1997. Our first public release was March 24, 1998.

Overall, Windows NT has been a good platform for protocol development. It accommodates new protocols, loadable at run-time, with great ease. The kernel debugger provides a good source-level debugging environment. Microsoft's Network Monitor tool, for capturing and viewing packets, was also very useful for debugging. We use Windows NT 4.0 for our development, but our MSR IPv6 implementation runs equally well on current versions of Windows NT 5.0.

We did come across several implementation challenges that are not specific to IPv6 and would be faced by any protocol implementation for Windows NT. Among the challenges for an efficient implementation are how to handle received packets given the multiplicity of ways to receive them, how to "pull-up" fragmented packet buffers into a contiguous buffer, and how to add link-layer headers when sending packets. In Section 4 we examine these and other problems and our solutions in detail.

We have had access to Windows NT source code during our development, which we have found useful but not essential. The learning curve for NT internals was very steep and sample code was particularly useful. (Our implementation can serve this purpose for others attempting Windows NT protocol development.) Not surprisingly, on several occasions source code was useful for debugging or for understanding poorly documented interfaces. In Section 5 we document more precisely when and how we made use of Windows NT source code.

The remainder of the paper is organized as follows. First, we present an overview of Windows NT's architecture for network protocols. Section 3 briefly describes our implementation and some of the major design choices we made. Next we report our experiences developing networking code for Windows NT, discussing solutions to problems inherent in Windows

NT's protocol architecture and also discussing the utility of access to Windows NT source code. Section 6 presents initial performance results for our implementation, and the next section compares our implementation to other IPv6 implementations. The paper ends with conclusions and a discussion of future work.

2. Windows NT Networking Architecture

Network protocols in Windows NT are dynamically loadable device drivers, much like any other device driver in Windows NT [1]. It is possible to add a new protocol to the system by writing two new components: a kernel-level driver (tcpip6.sys in Figure 1) that exports the TDI interface and uses the NDIS interface, and a user-level helper (wshipv6.dll in Figure 1) to support access to the driver via sockets.

Unlike the original BSD Unix sockets architecture, in which socket operations were direct system calls into the kernel, the Winsock architecture has several significant user-level components or Dynamic Link Libraries (DLLs). The Winsock DLL (ws2_32.dll) acts as a "traffic cop;" it redirects calls from the application to the appropriate Windows Socket Provider (WSP) or Name Space Provider (NSP). The WSP and NSP interfaces are both publicly documented. A WSP implementation can provide a new address family or an alternative implementation for an existing address family. WSP and NSP implementations make their presence known via entries in the system registry. The Microsoft WSP (msafd.dll) communicates with kernel drivers as described below, but other WSP implementations might function entirely at user-level or collaborate with a kernel driver via completely custom means. An NSP implementation supports name spaces for gethostbyname and related calls. For example, the Microsoft NSP (rnr20.dll) implements a DNS resolver.

To make it easier to add new protocols, msafd.dll supports multiple protocols through the use of helper DLLs. The helper DLL (like wshipv6.dll in our IPv6 implementation) exports the documented WSH interface, which msafd.dll uses when it wants to perform protocol-specific actions (like converting a socket address to a TDI address, or parsing the string representation of an address). msafd.dll handles almost all of the real work of being a Windows Socket Provider.

msafd.dll communicates with kernel protocol drivers via afd.sys. The interface between msafd.dll and afd.sys is not documented. Together msafd.dll and afd.sys handle buffering, select, and other minor issues as "glue" between Winsock and the TDI interface.

TDI (Transport-Device Independent) is Microsoft's kernel-level network protocol interface. It uses Microsoft's driver architecture [1], which encodes I/O operations in small structures called I/O Request Packets (IRPs). The driver architecture is inherently asynchronous. The kernel converts system calls into IRPs and drivers pass IRPs around until the operation that they represent completes. TDI is mostly just a family of operations encoded as complex ioctl IRPs. It uses its own address representation, distinct from Winsock's sockaddrs. It uses two kinds of pseudo-file objects, to represent connection-oriented and connectionless communication endpoints. It is documented, but not particularly well.

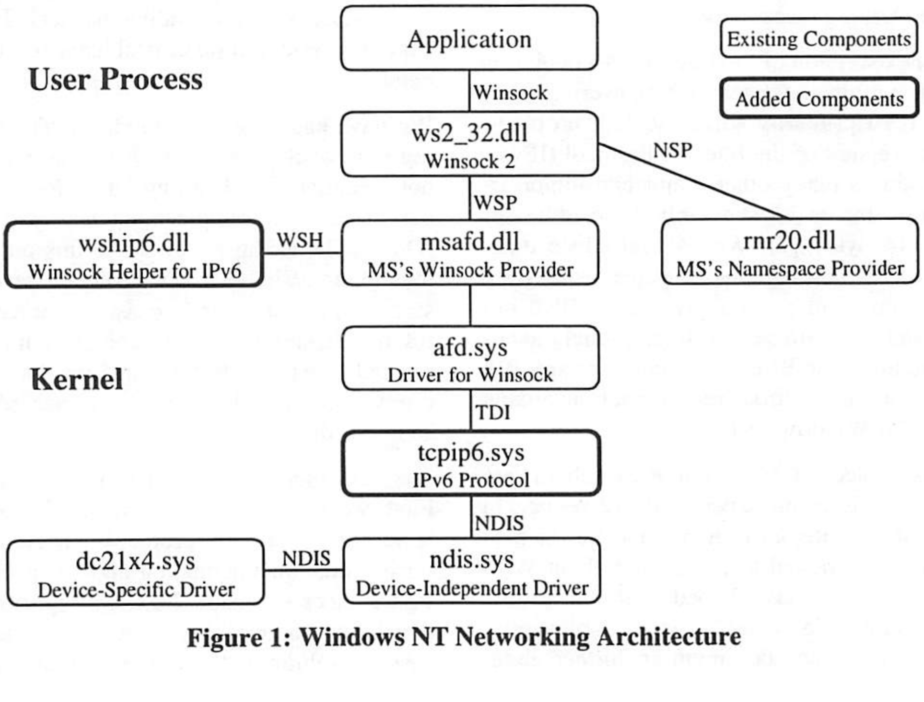


Figure 1: Windows NT Networking Architecture

A network protocol driver like tcpip6.sys implements TDI. Typically the protocol driver uses NDIS (Network Driver Interface Specification) to handle actual network interface cards. Like TDI, the NDIS interface is also asynchronous, but it uses callbacks instead of IRPs. For example the protocol calls NDIS to send a packet, and NDIS calls the protocol to indicate that a packet was sent and to indicate that a packet was received. Rather than have each hardware vendor reimplement NDIS, Microsoft supplies a common component (ndis.sys) that implements most NDIS functionality, and hardware vendors supply a relatively small "miniport" (like dc21x4.sys) that implements functionality specific to their device.

If one supplies an initialization file with configuration information, the standard Network Control Panel graphical user interface (GUI) can install, configure, and uninstall custom network protocols. The initialization file format is documented in the Device Driver Kit (DDK). Entries in the system registry control all configuration information, for network drivers, Winsock providers, and Winsock helpers. Registry entries also control the bindings between network interfaces and protocol stacks. For example, using Network Control Panel to modify the registry it is possible to disable a particular protocol for a given interface. One complication with configuration is that in Windows NT 5.0, both the initialization file format and the GUI for network installation and configuration have changed radically.

3. Our Implementation

Our implementation supports basic IPv6 functionality, but it is not a complete implementation. As of this writing, it supports Neighbor Discovery (which among other things replaces ARP), stateless address autoconfiguration (which allows hosts to configure automatically based on packets sent by routers), ICMPv6, Multicast Listener Discovery (essentially IGMPv2 for IPv6), several flavors of tunneling IPv6 via IPv4, and of course UDP and TCP over IPv6. It does not yet support security, authentication, mobility, or forwarding.

Future MSR IPv6 releases will include a separate driver for translating between IPv6 and IPv4, based on research at the University of Washington [4]. A translator allows an IPv6-only node to communicate with an IPv4-only node.

3.1. Implementation Strategy

We started with Windows NT 4.0 source for Microsoft's TCP/IP driver (tcpip.sys) and incrementally modified it. Eventually we replaced or rewrote all the IP-layer code, but our TCP and UDP layers are still strongly based on the original Microsoft code. Almost all of the machinery for supporting TDI is unchanged from the original code base. Starting with working IPv4 code and modifying it was very helpful in overcoming our learning curve and getting something working, but it resulted in intellectual property issues that we had to resolve when we wanted to plan a public source code release. Section 5 discusses our use of source code in more detail.

We also briefly considered starting with a public BSD-based IPv6 implementation and porting it to Windows NT. We feel that porting a BSD-based protocol, perhaps with TDI and NDIS glue layers, would be considerable work (the differences between BSD and Windows NT internals being much greater than the differences between IPv4 and IPv6) and probably result in an unsightly implementation. Because we would like our implementation to serve as a relatively clean example for others, we did not pursue this approach.

We decided to implement a "single stack," which only supports IPv6, as opposed to a "hybrid stack," which would support both IPv4 and IPv6 in an integrated fashion. We felt the single stack approach would be better for testing and experimentation, because the normal functions of the system, which rely on the IPv4 stack, are not affected by bugs or problems with the IPv6 stack. In practice this has worked very well and the presence of the experimental IPv6 stack has not caused problems for our systems. However for real product usage, the hybrid approach is probably superior for most scenarios. It eliminates the duplication of having separate IPv4 and IPv6 implementations of TCP and UDP. It also makes it easier to implement some transition mechanisms, like v4-mapped addresses (a way to have what seems to be an IPv6 socket which is really sending/receiving IPv4 packets) or an ioctl to change a IPv4 socket to IPv6 and back.

We decided very early on not to support Windows 95. The Windows 95 network architecture is similar to but not identical to Windows NT's, and we felt that for our purposes it was not worth the effort of understanding these differences, coding around them, and testing for two environments. The IPv4 code base from which we started was built with a proprietary glue layer to support both Windows NT and Windows 95, but we had to remove references to this glue layer prior to our public release. For a product implementation, Windows 95 support would be more interesting. The Windows 98 and Windows CE driver models more closely resemble Windows NT's, but we have not yet thoroughly explored the possibility of ports to those operating systems.

3.2. Code Organization

Our IPv6 implementation has three layers: the link layer, the network or IPv6 layer, and transport and other upper-layer protocols. (See Figure 2.) The IPv4 code base from which we started had the same divisions into three layers, but we simplified the interfaces between the layers.

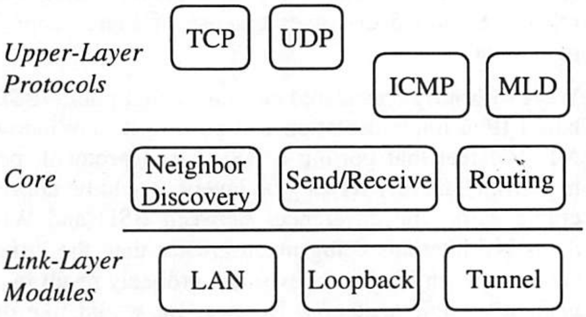


Figure 2: IPv6 Code Modules

The link-layer modules each manage a specific low-level interface type. The LAN module uses NDIS to handle Ethernet interfaces, and with minor changes it could also handle token-ring and other LAN media. The loopback module creates a pseudo-interface that reflects packets back to the sending machine. The tunnel module implements configured tunnels and automatic tunneling [5], and "6-over-4" [2]. In all three variations, the tunnel module uses IPv4 as the link layer to send and receive IPv6 packets. The tunnel module communicates with the IPv4 stack using TDI, in much the same way that the LAN module uses NDIS to communicate with Ethernet interfaces.

The core IPv6 modules communicate with link-layer modules through a well-defined interface. The link-layer module supplies IPv6 with a structure of information containing the length of link addresses, the network interface's own link-layer address, the link maximum transmission unit (MTU), and six entry points. The two substantial entry points send a packet and control multicast address assignment. The other four entry points are small helper functions, for creating an IPv6 address for the network interface given an IPv6 address prefix, for creating a link-layer multicast address given an IPv6 multicast address, and for reading and writing the link-layer address option fields in Neighbor Discovery packets. In the other direction, the link-layer modules call up to IPv6 to deliver received packets and to indicate the completion of packet-send operations. Our link-layer interface could easily be exposed to allow link-layer modules to reside in other kernel drivers or components, but we have not yet taken that step.

The IPv4 code base from which we started had a similarly well-defined link-layer interface, but the details were much more complicated. For example, IPv4 link-layer modules exported ten non-trivial entry points. Furthermore, IPv4 link-layer modules were responsible for allocating buffer space for their link-layer header (see Section 4.3) and for address resolution (ARP). In our implementation, common IPv6 code handles these responsibilities.

The core IPv6 modules implement the basic send/receive functionality, including fragmentation and reassembly and extension header processing, Neighbor Discovery (which replaces ARP for address resolution), and routing. Our current routing module only performs on-link determination and default router selection, using the data structures described below. We do not yet have any true routing table data structure or packet forwarding support.

The interface between higher-layer protocols like TCP or UDP and the core IPv6 code is fairly narrow but not yet as clearly defined as the link-layer interface. The IPv6 code calls up via a protocol switch table to deliver packets and control messages, and the higher-layer protocol code calls down to allocate packets, select source addresses, perform routing, and send packets. ICMP and MLD are technically upper-layer protocols but their implementation is integrated with the core IPv6 modules and data structures. For example, Neighbor Discovery uses ICMP messages and MLD uses the ADE data structures describe below.

3.3. Data Structures

Our design hews fairly closely to the conceptual data structures found in the Neighbor Discovery specification [11]. For example, it has a Neighbor Cache, a Destination Cache (although we call it a Route Cache), a Router List, and a Prefix List. The design deviates from the conceptual data structures in two major ways. First, we support multi-homed hosts (hosts with multiple interfaces), and this complicates the data structures slightly. Second, Route Cache entries cache the preferred source address for destinations as well as caching the next-hop neighbor, path MTU, and other information. Figure 3 presents the major data structures and their interconnections.

The Interface (IF) is our central data structure. There is one Interface for each network interface card, plus additional Interfaces for logical or virtual links like loopback, configured/automatic tunneling, and 6-over-4. In addition to link-layer information and configuration information, each Interface has a list of NTEs, which represent the addresses assigned to the interface that can be used as source addresses; a list of ADEs, which represent the addresses for which the Interface can receive packets; and a cache of NCEs, which represent neighboring nodes on that link.

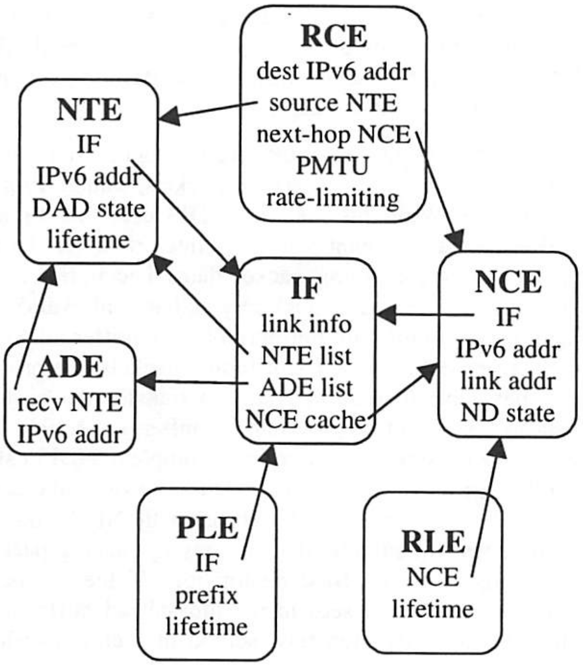


Figure 3: IPv6 Data Structures

The Net Table Entry (NTE) represents a unicast address assigned to an interface for use as a source address. It primarily contains state information for Duplicate Address Detection (DAD) and lifetime information derived from stateless address autoconfiguration. The name NTE derives from an IPv4 data structure similar in concept but different in detail (IPv4 does not have DAD or stateless autoconfiguration).

The Address Entry (ADE) represents a destination address for which the interface can receive packets. A unicast address can have both an ADE and an NTE, but a multicast address will have only an ADE. Each ADE maps to the NTE that logically receives the packets (would be used to reply to the packet if necessary). For example, IPv6 has the concept of addresses with link-local scope. An ADE for a multicast address with link-local scope maps to the NTE for the interface's link-local unicast address.

The Neighbor Cache Entry (NCE) represents a neighboring node on the interface's link. The NCE maps the neighbor's IPv6 address to its link-layer address. The Neighbor Discovery algorithms manage the state transitions for NCEs. We use a simple least-recently-used (LRU) algorithm to manage the cache.

The Prefix List Entry (PLE) and Router List Entry (RLE) represent routing information learned from advertisements sent by routers. Together they determine the next hop (NCE) to which a packet should be sent. If the destination address matches the prefix in a PLE, then the destination is assumed to be "on-link," or directly reachable. Otherwise the list of default routers (RLEs) must be consulted to choose a router for that destination.

The Route Cache Entry (RCE) caches the results of the next-hop selection algorithm. That is, an RCE maps a destination IPv6 address to a next-hop neighbor (NCE). In addition, the RCE caches the preferred source address (NTE) to use when sending to the destination, the path MTU, and ICMP error rate-limiting information. The name RCE derives from a similar IPv4 data structure.

We use a deliberately coarse-grained locking discipline for our data structures. Our intent is to refine the locking after we gain more experience with the data structures. Currently each Interface has a lock that protects the Interface itself and its NTEs, ADEs, and NCEs. There is a global route lock that protects PLEs, RLEs, and RCEs. In the normal case, sending a packet using a cached RCE does not require the acquisition of the route lock. The relevant Interface lock is briefly taken to send a packet (to check the Neighbor Discovery state) and to receive a packet (to search the ADE list).

The NCE, NTE, and RCE structures have reference counts, so pointers to them can be safely kept. Many of their interesting fields are read-only or can be safely accessed without holding a lock. The ADE, PLE, and RLE structures can only be accessed while holding the relevant Interface or route lock. The Interface structure does not have a reference count, but this is not a problem because our implementation does not yet support the plug-n-play features of Windows NT 5.0 that would allow interfaces to be removed at run-time. Once created, an Interface is never destroyed.

We also use deliberately simplistic data structures to represent our lists and caches. To keep the code simple, we use singly and doubly-linked lists instead of sorted arrays, trees, or hash tables. As we gain experience with the data structures in more demanding environments we plan to revisit these choices.

Using the conceptual data structures has been very successful for us. It has made it easy to track changes in the specification, because of the close correspondence between the spec and the code. We also hope that it will make our implementation more useful as an example for people interested in learning about IPv6. On the other hand, our implementation does not currently support routing tables, or forwarding of packets between interfaces. Efficient routing table support might require a more complicated data structure that would merge the PLE, RLE, and possibly the RCE data structures.

3.4. Configuration

Our implementation automatically configures itself as much as possible [14]. There is no configuration dialog in Network Control Panel. At boot time, it assigns link-local addresses to all interfaces, performs duplicate address detection, and solicits configuration information from routers. When it receives configuration information from a router, it automatically configures the receiving interface appropriately. This may involve creating new NTEs, PLEs, or RLEs. If the IPv4 stack is also present (normally the case), then it enables "automatic tunneling" [5] and creates a virtual "6-over-4" interface [2] for each IPv4 address. With automatic tunneling, packets sent to special v4-compatible IPv6 addresses are encapsulated and sent to an IPv4 destination address that is extracted from the low bits of the v4-compatible IPv6 address. With 6-over-4, the IPv6 code uses a multicast-capable IPv4 network as a true virtual link, with Neighbor Discovery and all other IPv6 features operational.

The only aspect of the implementation that must be configured manually, if the administrator wishes to enable it, is the "configured tunnels" transition mechanism [5]. For example, to connect an MSR IPv6 machine to the 6bone [12], a global IPv6 test network primarily composed of tunnels over the Internet, requires the following two registry entries. First, the administrator must supply an IPv6 address to be assigned to the tunnel interface. The machine accepts encapsulated IPv6 packets with this destination address. This results in the creation of an NTE and an ADE on the tunnel interface. Second, the administrator must supply the IPv4 address of the tunnel endpoint. IPv6 packets sent via the configured tunnel are encapsulated and sent to this IPv4 address. This results in the creation of an NCE (with a special state value that inhibits Neighbor Discovery) on the tunnel interface to represent the tunnel endpoint, and an RLE to represent the tunnel endpoint's role as a default router.

4. Problems and Solutions

During our implementation, we came across several challenges that would be faced by any protocol implementation for Windows NT. This section examines four such problems and our solutions in detail. In addition, we briefly document how our starting point IPv4 code addressed these problems. Note that we describe here the Windows NT 4.0 TCP/IP code base and Microsoft's TCP/IP stack has evolved very significantly in subsequent versions.

4.1. NDIS Receive Handlers

NDIS offers two different ways to receive packets. Unfortunately, the best method depends on the choice of network interface card and miniport. We were able to hide the differences between these methods inside the LAN link-layer module without sacrificing performance in any interesting cases.

A protocol implementation must support the most common method for receiving packets, called ProtocolReceive. With this method, NDIS calls the ProtocolReceive entry point with a pointer to a flat look-ahead buffer containing packet data. The buffer contents must be treated as read-only and are only valid for the duration of the call. Furthermore, the buffer may not contain the entire packet. For longer packets, the protocol may have to request NDIS to transfer the packet data to a new buffer (or chain of buffers) specified by the protocol. When this transfer is complete, NDIS calls another protocol entry point. This transfer-data case increases the interaction overhead with NDIS, but if used cleverly it can eliminate a copy by placing packet data directly in its final destination. If the protocol doesn't like what it sees in the look-ahead buffer and chooses not to transfer, this method may eliminate I/O operations.

NDIS 4.0 introduced an optional new method, called ProtocolReceivePacket, for receiving packets. (If a miniport wants to use ProtocolReceivePacket and the protocol does not support it, NDIS falls back to ProtocolReceive.) With this method, NDIS calls the ProtocolReceivePacket entry point with a pointer to a packet structure. The packet structure contains a chain of buffers, which the protocol must treat as read-only. The protocol can hold onto a received packet by returning a non-zero reference count to NDIS and later relinquish the references to return the packet. However it's not clear for how long a protocol may safely hold a packet. The miniport owns the packet's buffers, and preventing the miniport from reusing them may cause denial of service problems. Furthermore, in this situation the packet structure does not contain a "context" area that the protocol can use for its own purposes.

We examined the receive behavior of several different network interface cards and miniports. The Digital DE435 (dc21x4.sys) was the only one that used ProtocolReceivePacket. It always provided a packet with a single buffer. The SMC 9432TX EtherPower II (smc9432n.sys), the 3com 3c905 Fast Etherlink XL (el90x.sys), and the Intel EtherExpress PRO/10 (epro.sys) used ProtocolReceive, but the look-ahead buffer always contained the complete packet. The older Intel EtherExpress 16 (ee16.sys) was the only one that used ProtocolReceive with small look-ahead buffers, necessitating the use of transfer-data.

Given this data, we decided not to take advantage of ProtocolReceive's transfer-data case. We support transfer-data, so our implementation works with cards like the EtherExpress 16, but inside the link-layer module we always transfer-data immediately to a temporary buffer instead of postponing the transfer until the data's final destination is known. This hides the transfer-data complexity from the core IPv6 modules. However, it introduces a copy, relative to the IPv4 stack, in some circumstances.

We support ProtocolReceivePacket, but we do not take full advantage of its capabilities. Given the time scales involved, it seems ill-advised to hold onto the miniport's buffers across the reassembly of IPv6 fragments or to buffer data for TCP. Hence we always return to NDIS a zero reference count for the packet. When we implement packet forwarding, we may want to hold a packet from one interface just long enough to resend it on another interface; this will complicate our design slightly.

To present a common picture to the core IPv6 modules, our link-layer module hides the apparent differences between ProtocolReceive and ProtocolReceivePacket. Our receive path uses our own IPv6 packet structure, defined with the fields we want, instead of the NDIS packet structure. For ProtocolReceive, we initialize a stack-allocated IPv6 packet structure to point to the look-ahead buffer. In the ProtocolReceivePacket case, we initialize an IPv6 packet structure to point to the chain of buffers from the NDIS packet. Receiving modules can deal in a unified way with the IPv6 packet structure, reducing the number of parameters passed up the stack at each layer from seven to two.

The base IPv4 code did not support ProtocolReceivePacket. It implemented the transfer-data case of ProtocolReceive, but it would not defer the transfer-data in most interesting cases (like receiving a TCP packet). It took advantage of transfer-data to eliminate a copy when discarding packets not actually destined for the receiver, when reassembling fragments, and when forwarding packets.

4.2. Pull-up

Our implementation "pulls-up" non-contiguous packet data into contiguous data with no overhead in the common case when pull-up is not required. When examining headers, it is very convenient to have contiguous data, and in most cases the header is in fact contiguous. However, the IPv6 packet structure mentioned above does permit a packet representation consisting of a chain of buffers. In this situation, a header may fall across two or more buffers. A BSD-style pull-up design does not work, because the buffer chain is read-only for the protocol stack and must be left undisturbed.

With the network interface cards we have tried to date, the NDIS ProtocolReceivePacket indication never provides more than one buffer for a packet, rendering pull-up moot. However, the NDIS interface explicitly allows for multiple buffers and some cards may take advantage of this capability. Furthermore, our other link-layer modules do in practice deliver multi-buffer packets.

We use the IPv6 packet structure mentioned in Section 4.1 to solve the pull-up problem. To support this, the packet structure contains several relevant fields. The Data field points into the current data area being examined. The ContigSize field tracks the amount of remaining contiguous data, and the TotalSize field the total amount of data remaining. An Auxiliary field remembers any side allocation of buffer space for the most recent pull-up, so it can be properly freed. Normally the Data field points into the packet's first buffer. When a pull-up is required, auxiliary buffer space is allocated and initialized from two or more buffers in the chain and the Data field is updated to point to the auxiliary buffer area. This solution does not modify the original buffer chain, which is read-only for the protocol stack.

Our implementation leverages the common case checks to hide the pull-up if it is never required. Before casting a buffer data pointer to a header structure, a receive handler must in any case perform a length check to verify that there is enough remaining data. Figure 4 shows an example code fragment.

```
if (Packet->ContigSize < sizeof(Header))
    if (! PacketPullup(Packet, sizeof(Header)))
        ; // pullup failed - packet too small
h = (Header *) Packet->Data;
```

Figure 4: Pull-up Example

The base IPv4 code did not implement a generic pull-up solution. The IPv4 code did not support extension headers between the IP header and the upper-layer header, making it possible to bound the total amount of header data. Furthermore, the IPv4 code did not support ProtocolPacketReceive.

4.3. Adding Link-Layer Headers

Our implementation avoids chaining new buffers to add link-layer headers, reducing packet-send latency and simplifying the link-layer modules. When sending a packet, it is most efficient to allocate space for all the headers up front, as one contiguous area. However, when constructing a packet one does not always know in advance the exact total size of the headers. For example, the outgoing interface may be chosen later, and in that case the size of the link-layer header that will be needed is not known.

The obvious solution, which we use, is to track the maximum link-layer header size needed by any interface. When constructing the packet's headers, it is easy to leave space for the worst-case link-layer header at the front of the packet.

The problem with this solution is that the NDIS packet structure does not allow for unused buffer space at the front of a packet. This will occur if the actual NDIS interface has a smaller link-layer header than the pessimistically allocated maximum size. The same problem occurs when sending a packet via TDI in the tunnel link-layer module.

To solve this problem, we make a pair of helper routines available to link-layer modules. The first helper routine rewrites a packet's first buffer descriptor (which is actually a memory descriptor list, or MDL), to hide any unused buffer space. The second helper routine undoes the work of the first, so that the link-layer module can return the packet unchanged. The two helper routines communicate some state information (an offset value) via the protocol context area of the NDIS packet structure. The NDIS packet format allows for a context area, or protocol-defined annex following the main packet structure. The only tricky aspect to rewriting the MDL occurs if the unused buffer space crosses a physical page boundary, in which case an array of physical page addresses in the MDL must be rewritten in place.

4.4. Preventing Deadlock

During our implementation effort, we discovered that our IPv6 stack would hang some machines during boot. Debugging this problem (see Section 5.2) led us to an interesting deadlock between the core IPv6 modules and our link-layer modules. This section explains the deadlock, which is a potential danger for any protocol implementation, and our solution for avoiding it.

The potential for deadlock lies in NDIS's control operations. NDIS supports control operations such as changing the current list of link-layer multicast addresses to which the interface should listen. These control operations are asynchronous: one calls NDIS to request the operation, and NDIS calls a protocol entry point to indicate completion. This is inconvenient, and in fact our LAN link-layer module has a helper function (inherited from the IPv4 code base) that implements a synchronous control operation. The helper function initiates the control operation and then waits for the completion call back via a synchronization object.

The problem arose when our stateless address autoconfiguration code would configure a new address on an interface. This could change the list of multicast addresses for the interface, because for each IPv6 unicast address there is a corresponding address, the solicited-node multicast address, used in Neighbor Discovery. A change in the list of IPv6 multicast addresses in turn could change the link-layer multicast addresses. This would result in a "synchronous" call into NDIS via the helper function. In most cases this worked fine, but with some interface cards it would deadlock because NDIS would call the protocol stack's receive entry point while holding an internal lock. The multicast address control operation needed to acquire this same lock.

We tried two different solutions. The first solution was to expose in our internal link-layer interface the inherently asynchronous nature of the set-multicast-address-list control operation. This worked, but it greatly complicated our link-layer interface. The final solution was to document that the set-multicast-address-list operation in the link-layer interface can only be called from a safe thread context; it can not be called from the receive path. When the receive path wants to invoke this operation (for example because of stateless address autoconfiguration), the actual call is deferred to a kernel worker thread.

The final solution also solved a problem with the tunnel link-layer module. The tunnel link-layer module uses the TDI interface to the IPv4 stack instead of NDIS. The IPv4 stack exposes a TDI operation for controlling multicast addresses. We never saw this problem in practice, but we realized that the IPv4 implementation of this TDI operation implicitly assumed that it was being called from a preemptible thread context. (For example, it executes pageable code.)

The lesson we draw from this is to carefully document the call-context assumptions in interfaces. It should be clear whether a function must be called from a preemptible thread context, what locks may be held when a function is called, etc. The NDIS and TDI interfaces are both inadequately documented from this perspective.

5. Source Code Access

In this section, we explore how we made use of Windows NT source code. We have access to all Windows NT source, and in fact for our MSR IPv6 implementation we started from the source code for TCP/IP in Windows NT 4.0. The TCP/IP stack was very valuable as sample code, but in the end we have replaced almost all of the core IP-layer code. We have made limited use of the source for other Windows NT components.

5.1. Source for Windows NT 4.0 TCP/IP

We based our implementation on an old version (NT 4.0 with no Service Packs) of Microsoft's IPv4 protocol stack. This source code played several roles for us:

- Sample code. Sample source code was essential for us to get started quickly. We had no prior experience with network protocol programming for Windows NT.

- UDP/TCP code. Not having to implement UDP, and especially, TCP has saved us much effort.
- TDI glue code. Our stack's support for the TDI interface derives almost unchanged from the IPv4 code. This includes the code for managing TDI's pseudo-file objects that represent communication endpoints.

At this point, we have replaced or written from scratch most lines of code in the link-layer modules, the core IPv6 modules, and ICMP and MLD. There is still a noticeable genetic resemblance to the original IPv4 code, in terms of code organization and function and variable names. In the UDP and TCP modules we have made relatively minor changes. We changed the buffer handling in TCP to support our IPv6 packet structure in the receive path. We updated the UDP and TCP checksum calculations. We updated variables, fields, and parameters that represented addresses. Addresses as parameters generally changed from by-value to by-reference.

For sample code, we did not look early enough at Microsoft's Driver Development Kit (DDK). The DDK has a sample network driver, which looks useful but we have not evaluated it carefully. The DDK also contains a sample Winsock helper DLL. This sample is really just the Windows NT 4.0 IPv4 helper, with minor modifications for the DDK build environment. Our IPv6 Winsock helper DLL was originally based on Windows NT source, but we easily recreated it based on the DDK sample. More importantly, the Winsock helper sample implicitly reveals some aspects of the TDI interface to the IPv4 stack that are nowhere else documented. For example, our tunnel code uses the IPv4 stack via TDI and the ioctls (specific to the IPv4 stack) for controlling multicast are only documented in this DDK sample.

5.2. Other Windows NT Components

We have made limited use of the source code for other Windows NT components. It has occasionally been useful for debugging or for informational purposes, when interfaces were inadequately documented. We have not modified any existing Windows NT components for our implementation, with the exception of one bug fix in msafd.dll.

We did not have good sample code for Winsock's WSALookupServiceBegin/Next/End APIs, and source code for rnr20.dll (the DNS Name Space Provider) was essential for our use of these ostensibly documented APIs. These APIs are a generalized version of gethostbyname. In fact it turns out that with the right combination of arguments, one can use these APIs to request AAAA (IPv6 address) records from DNS, and have the raw DNS replies returned. We have wrapped this in a helper function that looks up IPv6 addresses, despite the fact that rnr20.dll does not support IPv6. Unfortunately in this usage, rnr20.dll does not cache the DNS replies.

Source code for other Windows NT components (like ntoskrnl.exe, afd.sys, ndis.sys) was occasionally very useful for debugging but not essential to the project. During debugging it would be helpful to step through these other components as well as the IPv6 driver to understand a problem. The deadlock problem described in Section 4.4 was a prime example of this.

We came across a problem in getsockname that we traced to a bug in msafd.dll that only showed up with large addresses. Of course, source code for the Winsock components was essential for finding and fixing this.

5.3. Network Monitor

Microsoft's Network Monitor tool captures network packets and parses them for display. It has been extremely useful during our development. It supports run-time loadable parsers, developed with Microsoft's SDK. The parser architecture is fairly simple and clean and it was very easy to write a new IPv6 parser. Although in general new parsers can be developed without access to Network Monitor source, we did have to modify the existing IPv4, UDP, and TCP parsers to deal properly with the interactions between the IP layer and the UDP/TCP layers. One example is verifying the TCP checksum in the TCP parser, when the previous header could be IPv4 or IPv6.

6. Performance

We examined the TCP performance of our IPv6 implementation, and found roughly 2% performance degradation relative to IPv4 for both 10 and 100 Mb/s Ethernet. Because IPv6 packets have a larger header, we expected roughly 1.4% performance degradation. We have not yet tuned our stack for performance, so we are satisfied with these initial results. Fiuczynski [4] has performance numbers for an earlier version of our implementation.

Our testing environment consisted of two machines directly connected via a reversing Ethernet cable. The sending machine was a 300 MHz Pentium II Gateway E5000; the receiving machine was a 266 MHz Pentium II Gateway E3100. Both were equipped with SMC 9432TX EtherPower II network interface cards, which are capable of running in either 10 or 100 Mb/s mode. We performed tests in both modes. While the cards also support full-duplex transfers, we performed all our testing at the more commonly used half-duplex setting.

Both machines were running Windows NT Version 4.0 with Service Pack 3 installed. However, to achieve a fair comparison we replaced the TCP/IPv4 driver (tcpip.sys) with a driver that we built using the Windows NT 4.0 sources from which we derived our IPv6 implementation. The software driving the test was the public domain "ttcp" program, which we ported to run on our IPv6 stack. We used 128KB socket buffers. (We observed very similar performance with 64KB buffers, but buffers smaller than 64KB performed poorer.)

We ran TCP throughput tests for both IPv4 and IPv6 over both 10 Mb/s and 100 Mb/s Ethernet. Each test was run six times, with each run taking about 100 seconds. We present the mean throughput and standard deviation for each test.

We expected to see a performance degradation of approximately 1.4% with the IPv6 stack. IPv6 headers are 40 bytes, IPv4 headers are 20 bytes, and TCP headers are 20 bytes. The maximum Ethernet frame size is 1500 bytes. So the degradation is 20 bytes out of 1460 bytes.

	10 Mb/s	100 Mb/s
IPv4	1058±4	10995±20
IPv6	1032±3	10790±30

Table 1: TCP Throughput in KB/s

Table 1 depicts our actual results. For 10 Mb/s Ethernet, we see 2.5% degradation. For 100 Mb/s Ethernet, we see 1.9% degradation.

While performing these measurements, we noticed several things that we have not been able to follow up on. The IPv4 stack that we built yields slightly better throughput than the IPv4 stack released with Windows NT 4.0 Service Pack 3. With the 266 MHz machine sending and the 300 MHz machine receiving, we observed better 10 Mb/s performance and poorer 100 Mb/s performance. We also tried Digital DE500 interface cards, but we observed substantially poorer 100 Mb/s performance.

7. Other IPv6 Implementations

This section describes our IPv6 stack in comparison with other current IPv6 implementations.

The IPng Home Page [9] implementations list includes many implementations for host computers, notably actively supported "early adopter" implementations from Digital [6] and Sun [13]. These implementations are not available as source code.

Both Digital and Sun developed hybrid stack implementations, in which an integrated implementation of IPv4 and IPv6 share common transport-layer code. An IPv6 socket (AF_INET6) is usable for both IPv6 and IPv4 traffic, so that the result of a domain name lookup at runtime can determine what IP should be used on output, and the arrival of an IPv6 or IPv4 datagram determines the stack used on input.

We can compare the Digital and Sun implementations with MSR IPv6 and another freely distributed source code, that of INRIA Roquencourt [8]. We and INRIA have separate transport modules for IPv6. In our case, the goal was to avoid interfering with IPv4 traffic in any way during experiments using IPv6. MSR IPv6 does not modify the TCP/IPv4 driver. The drawback is excess code in the system and a requirement for two sockets in applications for them to run over both IP versions. This seems like the right tradeoff for a frequently changing research environment where both the stack and the applications may be modified at will.

On other dimensions, the four stacks here are very comparable: all have complete implementations of the base specifications and Neighbor Discovery. None have as yet fully implemented the required IPSec and mobility functions. As of this writing, compared to the other implementations MSR IPv6 has more complete multicast support.

7.1. Size Comparison with INRIA IPv6

This section compares the MSR IPv6 source code with the freely available INRIA implementation's source code [8].

INRIA used the strategy of creating clones of the IPv4 files for IPv6. INRIA IPv6 files are very similar to their IPv4 counterparts. In contrast, though MSR IPv6 started from IPv4 code, the core IPv6 modules have been largely designed and written from scratch.

Table 2 shows code size comparisons of INRIA IPv6 and IPv4 and of MSR IPv6 and its IPv4 starting point code. The table uses raw source lines because it compares INRIA to INRIA and MSR to MSR, and a partial set of comparisons with comments stripped looked very similar.

Module	INRIAv4	INRIAv6	MSRv4	MSRv6
IP input	1427	2205	1145	1109[1]
IP output	1432	2503	1867	451[1]
TCP	4239	4678	12089	11275
ICMP[2]	1653	1654	2286	2230

Table 2: Lines of Code

[1] MSR IPv6's subr.c (869 lines) is not included in the IP input and output line counts. [2] Including IGMP or MLD, but not Neighbor Discovery.

As expected, the cloning method results in larger code sizes for adding IPv6 to an IPv4 base. The method of rewriting and optimizing that we did often results in a code for an IPv6 function, such as packet output, that is smaller than the parallel IPv4 function. To a degree that is difficult to quantify, the streamlined design of the IPv6 base protocol may contribute to the shrinkage. For example, IPv6 has simpler header structures.

The INRIA ICMP module has similar line counts for IPv4 and IPv6 because the revisions in the design of ICMP from IPv4 to IPv6 have led INRIA to rewrite this code instead of cloning it.

The minimal effect of IPv6 on TCP is clear from the comparison of TCPv4 and TCPv6 code sizes. The TCP pseudo-header checksum calculation changes for IPv6, but otherwise the protocol is not modified. The shrinkage seen for the MSR TCP results from simplifications to buffer management stemming from our IPv6 packet structure. The MSR TCP implementation does not yet take advantage of an opportunity to optimize Neighbor Discovery by using reachability information gleaned from TCP acknowledgments to suppress Neighbor Discovery messages. This would add slightly to the TCP code size for IPv6.

8. Conclusions

Once past the learning curve, we have found Windows NT to be a good platform for protocol development. The internal interfaces are often complex, but so far we have been able to accomplish everything we need to do with them.

We plan to continue our development, with periodic public source code releases. Our first public source release was March 24, 1998, available at http://www.research.microsoft.com/msripv6. Our first priority in subsequent releases is finishing a full host implementation, including security, authentication, and mobility support, and adding interesting applications to the release.

Beyond our own work of completing this IPv6 implementation, we believe also that the code is a good resource for hands-on research in a variety of areas. Some examples include active network protocols used beside or instead of IP, new unicast and multicast transport protocols or algorithms, signaling protocols, queue management algorithms, and network aware applications.

Acknowledgments

Maryann Pérez Maher and Paul Dyke contributed significantly to our implementation. Marc Fiuczynski wrote our IPv6-IPv4 translator, based on his research at the University of Washington. The Windows Networking Group generously allowed us to release portions of their code in our public distribution.

References

[1] Art Baker. The Windows NT Device Driver Book: A Guide for Programmers. Prentice-Hall PTR, Upper Saddle River, New Jersey, 1997.

[2] B. Carpenter and C. Jung. Transmission of IPv6 Packets over IPv4 Networks without Tunnels. Internet Draft, draft-carpenter-ipng-6over4-03.txt, September 1997.

[3] S. Deering, R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 1883, December 1995.

[4] Marc E. Fiuczynski, Vincent K. Lam, and Brian N. Bershad. The Design and Implementation of an IPv6/IPv4 Network Address and Protocol Translator. Proceedings of the 1998 USENIX Technical Conference, June 1998.

[5] R. Gilligan, E. Nordmark. Transition Mechanisms for IPv6 Hosts and Routers. RFC 1933, April 1996.

[6] Daniel T. Harrington, James P. Bound, John J. McCann, Matt Thomas. Internet Protocol Version 6 and the Digital UNIX Implementation Experience. Digital Technical Journal, Volume 8, Number 3, http://www.digital.com/DTJN01/DTJN01HM.HTM, 1996.

[7] R. Hinden, S. Deering. IP Version 6 Addressing Architecture. RFC 1884, December 1995.

[8] INRIA Rocquencourt IPv6. ftp://ftp.inria.fr/network/ipv6/.

[9] IPng Working Group Web Site. http://playground.sun.com/pub/ipng/html/ipng-main.html.

[10] Microsoft Research IPv6 Implementation. http://www.research.microsoft.com/msripv6.

[11] T. Narten, E. Nordmark, W. Simpson. Neighbor Discovery for IP Version 6 (IPv6). RFC 1970, August 1996.

[12] 6bone Web Site. http://www.6bone.net/.

[13] IPv6 for Solaris. http://playground.sun.com/pub/solaris2-ipv6/html/solaris2-ipv6.html.

[14] S. Thompson, T. Narten. IPv6 Stateless Address Autoconfiguration. RFC 1971, August 1996.

A Soft Real-time Scheduling Server on the Windows NT

Chih-han Lin, Hao-hua Chu, Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana Champaign
clin2, h-chu3, klara@cs.uiuc.edu

Abstract

We present the design and implementation of a soft real time CPU server for the time-sensitive multimedia applications in the Windows NT environment. The server is a *user-level* daemon process from which multimedia applications can request and acquire periodic processing time in the well-known form of (*processing time* per *period*). Our server is based on a careful manipulation of the real time(RT) priority class, and it does not require any modifications to the kernel. It provides (1) the rate monotonic scheduling algorithm, (2) support for multiple processors (SMP model), (3) limited overrun protection among real-time(RT) processes, (4) fair allocation between the RT and time sharing (TS) processes so that TS processes are not starved for processing time, (5) accessibility by a normal user privilege, and (6) an efficient implementation. We have implemented the CPU scheduling server on top of the Windows NT 4.0 operating system with dual Pentium processors, and we have shown through experiments that our CPU scheduling server provides good soft real time support for the multimedia applications.

1. Introduction

Continuous media processing, such as video/audio compression/decompression, and 3-D rendering and animation, are becoming widely-used applications on computers nowadays. To preserve their temporal behavior, multimedia applications require that the underlying systems provide sufficient and periodic processing time and enforce quality guarantees to the users (e.g. a fixed video playback rate). However, in the Windows NT multi-process and time-sharing environment, these multimedia applications do not perform well when they are scheduled concurrently with the traditional TS applications such as text editors, compilers, web browsers, or computation-intensive jobs. Oftentimes, the problem lies in untimely scheduling of the processes rather than insufficient processor capacity. This paper addresses this problem and presents a user-level middleware solution on top of the Windows NT operating system with multiple processors.

There have been several research results that address the issues of accommodating scheduling of soft RT processes in general purpose operating system environment. They are the Constant Utilization Servers[2], the Processor Reserve of the RT Mach[8,6], the Hierarchical CPU scheduler[4], the User-level Real Time Scheduler[7], the Real Time Upcall[3], the Rate-Controlled Scheduling[11], the Soft RT scheduling Server[9], the Rialto operating system[5], the Nemesis[1], and the SMART system[10]. All except [10] are based on the general concepts of *reservation, resource allocation, and scheduling.* The RT process first sends a reservation request, which specifies its resource demand, e.g., RT Mach convention of (requested CPU usage time, period), to the resource manager. Then the resource manager performs an admission control to determine if there is enough available resource to allocate for this request. If the admission control test succeeds, the RT process is scheduled according to the reserv ation contract.

Our scheduling mechanism is based on the *user-level RT scheduler(URSched)* proposed by Kamada[7] in UNIX. The URSched mechanism is based on the POSIX.4 fixed priority extension and its priority scheduling rule. The user-level scheduler is implemented on top of the kernel scheduler, and it runs at the highest possible fixed-priority. The RT process waits its turn at the lowest possible priority (called the waiting priority), and the active RT processes run at the 2nd highest fixed priority. The user-level scheduler wakes up periodically to dispatch the RT processes by moving them between the waiting and the running priority; during the other time, it just sleeps. When the scheduler sleeps, the RT process executes at the running priority. When no RT processes are dispatched, the TS processes with dynamic priorities are scheduled by the underlying kernel scheduler. This approach has shown to have many desirable prope rties:

- It requires no modification to the kernels. The RT scheduler is implemented as a user-level process.
- It has low computation overhead.
- It provides the flexibility to implement any scheduling algorithms in the user-level scheduler, e.g., rate monotonic, earliest deadline first, or a hierarchical scheduler.

The above discussed scheduling mechanism was used and further expanded by additional mechanisms, algorithms, and policies in our QoS-aware resource management middleware [9], called QualMan. In this context, the middleware is understood as a system software between operating system and applications that provides access to extended and flexible OS services, e.g. real time support, to applications without any modifications of the operating systems. Our middleware provides the following ser vices :

- Rate Monotonic Scheduling algorithm.
- Overrun Protection among RT processes.
- Fair allocation between the RT and TS processes.
- Access to system services with normal user privileges.

Based on lessons learned from the soft RT scheduling server in the UNIX environment, we design, implement, and test the soft real-time scheduling server in the Windows NT environment. In addition, we provide support for scheduling of RT and TS processes on multiple processors. The paper is organized as follows: section 2 explains the scheduling server architecture; section 3 describes the implementation on the Windows NT platform, and discusses the differences between UNIX and the Windows NT operating system; section 4 shows the experimental results; section 5 presents the concluding remarks.

2. Server Architecture and Design

Our server architecture contains three major components—the broker, the dispatcher, and the dispatch tables as shown in Figure 1. A RT Client is an external component representing an application which requests the scheduling services from the scheduling server.

Before we describe each component in detail, we discuss the priority levels in the Windows NT system because they play an important role in our architecture and design. Each NT process or its main thread has a scheduling priority. Each thread's priority is determined by the priority class of its process and the priority level of the thread within the priority class of its process. Note that our scheduler is a *process* scheduler and it does not schedule the various threads in each RT process. There are four possible priority classes for a process:

1	IDLE_PRIORITY_CLASS
2	NORMAL_PRIORITY_CLASS
3	HIGH_PRIORITY_CLASS
4	REALTIME_PRIORITY_CLASS

There are seven possible priority levels within each priority class:

1	THREAD_PRIORITY_IDLE
2	THREAD_PRIORITY_LOWEST
3	THREAD_PRIORITY_BELOW_NORMAL
4	THREAD_PRIORITY_NORMAL
5	THREAD_PRIORITY_ABOVE_NORMAL
6	THREAD_PRIORITY_HIGHEST
7	THREAD_PRIORITY_TIME_CRITICAL

2.1 Client RT Process

A RT process can reserve a certain amount of CPU time from the real-time scheduling server. At a later time, it can also free or modify a reservation through an application program interface (API) defined in Table 1.

The client's reservation request contains the process ID and its resource specification in the form: period=P(ms), and CPU usage in percentage=U. The amount of CPU time per period, denoted E, is computed as $E = U*P$. For example, if a RT process requests a reservation for CPU usage = 30%, period = 100ms, and the request is accepted by the scheduling server, then the scheduling server will guarantee that the RT process is scheduled for 30ms of CPU time every 100ms, given that the RT process is runnable.

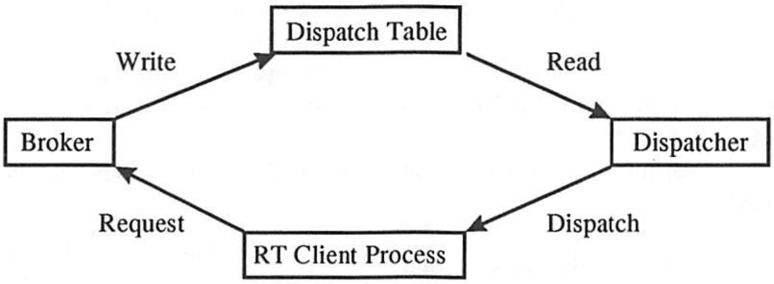


Figure 1: The Soft RT Server Architecture

Table 1: Application program interface

int Cpu::reserve(int *pid*, double *util*, int *period*)	Reserve CPU time corresponding to *util* over the time *period* for the process *pid*. Return 1/0 for resource admission su ccess/failure.
int Cpu::freeReserve(int *pid*)	Free the reservation hold by process *pid*. Return 1/0 for success/failure.
double Cpu::getAvailResource()	Return the amount of available CPU resource in the system.
int Cpu::modifyReserve(int *pid*, double *newUtil*, int *newPeriod*)	Modify the reservation held by process *pid*. Return 1/0 for resource admission success/failure.

The users start the RT process at the NORMAL_PRIORITY_CLASS which is like any other Time Sharing (TS) processes. When the RT process calls the CPU reserve() API, it opens an inter-process communication (IPC) with the broker through which the reserve parameters and the acceptance result are exchanged. To address the security problem which one process can modify or free another process's resource reservation through the API, the broker performs an ownership check to make sure that the calling process can be permitted to change the resource reservation of the specified RT process. Then the broker and the dispatcher schedule the RT process by manipulating its priority, the mechanism is described below.

2.2 Broker and Dispatch Table

The broker receives requests from the client RT processes. It performs an admission control test for the non-preemptive rate monotonic scheduling algorithm [12] to determine whether the new client RT process can be scheduled. Note that all equations must be satisfied for the admission test to succeed. Given a total of *m* periodic RT processes and a single processor system (*N*=1), let the RT processes be sorted according to sizes of their periods. Let e_i and p_i be the execution time and the period of the i-th client RT process, e_m and p_m be the execution time and period of the RT process with the smallest period, and *r* be the overall percentage of processor capacity allocated to the RT processes.

$$\sum_{i=1}^{m} \frac{e_i}{p_i} \le r N \quad \textbf{(eq. 1)}$$

$$p_m \ge e_m + max_{(1 \le i < m)} e_i \quad \textbf{(eq. 2)}$$

$$p_i \ge e_i + max_{(1 \le j \le m,\, j \ne i)} e_j + \sum_{j=i+1}^{m} e_j F(p_i - e_j, p_j) \quad \textbf{(eq. 3)}$$

$$where F(x, y) = ceil(\frac{x}{y}) + 1$$

Given an *N* processors system, the broker needs to decide how to place the multiple RT processes into the multiple processors. The broker maintains a set of RT processes that are admitted and are assigned to run on each of the processors. The set of processes for each processor must satisfy both **eq. 2** and 3. When a new RT process request arrives, the broker tries to place the new RT process in each of the processors *(1..N)* by performing the above one-processor admission trial on the processor. During the one-processor admission trial, the new process is inserted into the existing set of processes corresponding to that processor, and **eq. 2** and **3** are checked. If the trial succeeds, the broker will admit the new RT process which is assigned to that processor. If the trial fails, the broker will try to place the new RT process in the next processor. If the broker cannot find any processor that can accommodate the request, the RT process is rejected and not schedulable.

If it is schedulable, the broker will put the RT process into the waiting RT process pool by changing its priority to the waiting priority at IDLE_PRIORITY_CLASS level.

The broker is a daemon process running at a normal priority (The server's priority class is NORMAL_PRIORITY_CLASS and the priority of its primary thread is THREAD_PRIORITY_NORMAL). It can be started at the system boot time. It will wake up when a new client RT process request arrives. The broker needs to be run with the privilege of LocalSystem (equivalent to the root privilege in UNIX) so that it can start a RT process. The broker process does not perform the actual dispatching of the RT processes and does not enforce reservations. However, it needs to start (at the startup time) the dispatching process. The reason for separation between the broker process and the dispatcher process is that the admission and schedulability test in the broker may have variable computation time, hence it may affect the timing of dispatching. The other reason is that the admission and schedulability tests do not need to be done in real time. As a result, the broker runs at a normal priority and the dispatcher at a higher RT priority.

The broker computes its schedule for the RT processes in its dispatch table using the non-preemptive rate monotonic algorithm. The dispatch table is a shared memory object where the broker writes the computed schedule into it and the dispatcher reads from it. The dispatch table contains a repeatable time frame of slots, each slot corresponds to a time slice of a processor. Each slot can be assigned to a RT process PID, or is free which means yielding the control to the NT kernel scheduler to schedule TS processes. Note that Windows NT allows the system to have multiple processors in a symmetric multiprocessing model (SMP). Given N processors, we can run N processes concurrently. Therefore the dispatch table has N columns of repeatable time slots, where the i-th column corresponds to the schedule on the i-th processor. We show a sample dispatch table for a dual processors system in Table 2. Note that it is possible that the broker may make a massive change to the dispatch table when accepting a new RT process, while the dispatcher is dispatching time slots located at middle of a time frame. This massive change may cause undesirable shifts in the rate monotonic schedule and may disrupt the guaranteed time slots assigned to some RT processes. As a result, the dispatcher will keep a separate private copy of the dispatch table which it uses for dispatching. This private copy of the dispatch table is only updated with the broker when the dispatcher reaches the end of its time frame.

Table 2: A sample dispatch table for a dual processors system.

Slot Number	Time	Process PID	Process PID
0	0-20ms	100	102
1	20-40ms	101	103
2	40-60ms	100	102
3	60-80ms	free	free
4	80-100ms	100	102
5	100-120ms	free	103
6	120-140ms	100	102
7	140-160ms	free	free

The repeatable frame in Table 2 has a length of 160ms, and it contains 8 time slots of 20ms each. The sample dispatch table is a result of a rate monotonic schedule, where process 100 is assigned to run on processor #1 and according to the contract (period=40ms, execution time=20ms), process 101 is assigned to run on processor #1 and according to the contract (period=160ms, execution time=20ms), process 102 is assigned to run on processor #2 and according to the contract (period=40ms, execution time=20ms), and processor 103 is assigned to run on processor #2 and according to the contract (period=80ms, execution time=20ms). There are 4 free slots where TS processes can run. The minimum number of free slots is maintained by the broker to provide a fair share of the CPU time to the TS processes. In the above dispatch table, a total of 31.25% (100ms out of possible 320ms) of processing capacity is guaranteed to the TS processes. The site administrator can adjust this TS allocation value, which is (1-r) in the admission control equations, to be what is considered as a fair allocation between the TS and RT processes. For example, if the computer is used heavily for RT applications, the TS allocation value can be set to a small percentage number.

2.3 Dispatcher

The dispatcher is a periodic process running at the highest possible fixed priority with (REALTIME_PRIORITY_CLASS +THREAD_PRIORITY_TIME_CRITICAL). The dispatcher maps the dispatch table into its address space, and its job is to dispatch the RT process recorded at the time slot for all the processors. The dispatcher contains the next slot number. At the beginning of each dispatch slot, a periodic RT timer signals the dispatcher to schedule the next RT process. The length of time to switch from the end of one time slot to the beginning of the next time slot is called the dispatch latency. The dispatch latency is our scheduling overhead and it should be kept at a minimal value.

Consider the sample dispatch table in Table 2 at 20 ms when the next slot number is 1. The following steps are taken by the dispatcher:

1. The periodic timer wakes up the dispatcher process at 20ms. The dispatcher preempts RT process 100(or 102) if it is running on processor 1(or 2).

2. The dispatcher sets the RT processes 100 and 102 to the waiting priority (IDLE_PRIORITY_CLASS) using the system call SetPriorityClass().

3. The dispatcher promotes the RT processes 101 and 103 to the running priority (REALTIME_PRIORITY_CLASS). It binds the RT processes 101/102 to processors #1/#2 using the system call SetProcessAffinityMask().

4. The dispatcher puts itself to sleep. As a result, the RT processes 101/102 are scheduled on processors #1/#2 for 20ms until the timer wakes up the dispatcher again.

We have encountered a problem in the Windows NT that the periodic timer may fail to wake up the dispatcher when the RT process overruns its assigned slot. Overrun is defined as an additional time to the reserved processing time of an admitted RT process. Take the example of process 101 in Table 2 and it has reserved 20ms out of every 160ms. It would be overrunning when it uses more than 20 ms of processing time (say 30ms) to complete one iteration run. When an overrun occurs, the invocation of the dispatcher will be delayed till the RT process finishes its overrun due to the timer problem. This means that one process overrun can delay the dispatching of the process in the next time slot. Hence, until we resolve the preemption timer problem in Windows NT, we assume well-behaved RT processes. In the meantime, the dispatcher takes two actions to control the overrun. The first one is that the dispatcher will monitor for any *misbehaving* RT processes that are overrunning on a regular basis and it will remove them. We currently define a misbehaving process as one that is overrunning for more than 20% of its reservation and for more than 3 iterations during the most recent 10 iterations. Take the example of process 101 in Table 2. It would be misbehaving if during the 10 most recent iteration runs, it uses more than 24ms (which is 20% more than the 20ms reservation) of processing time for more than 3 times. The parameters that define a misbehaving process can be set by the system administration to be either more strict or lax. The second action is that the dispatcher will use some of the TS allocation, by temporarily assigning the free slots to the RT process whose time slots are taken by overrunning RT processes.

3. Implementation

We have implemented our server architecture on a HP Vectra Xu system which has two Intel Pentium 200 processors and 96M of memory. It runs Windows NT 4.0 operating system. The dispatch latency consists of 4 SetPriorityClass() system calls, 2 SetProcessAffinityMask() system calls, and 3 context switches. The average dispatch latency, over 10,000 runs, is measured as 0.64 ms. The time slot is set to be 20ms, and the overhead comes to be 3.2%.

4. Experimental Result

The experiment consists of the following load of applications running concurrently:

- The MPEG player, written by MSSG (MPEG Software Simulation Group), plays a 320x240 MPEG-1 file “twister.mpg” that contains a clip of the movie Twister at 20 frames per second (FPS).
- The same MPEG player plays a second 320x240 MPEG-1 file “lecture.mpg” that contains a speaker giving a lecture, at 20 frames per second.
- The Microsoft Visual C++ Compiler compiles the MSSG MPEG player.
- Four compute programs calculate the sin and cos tables using the infinite series formula.

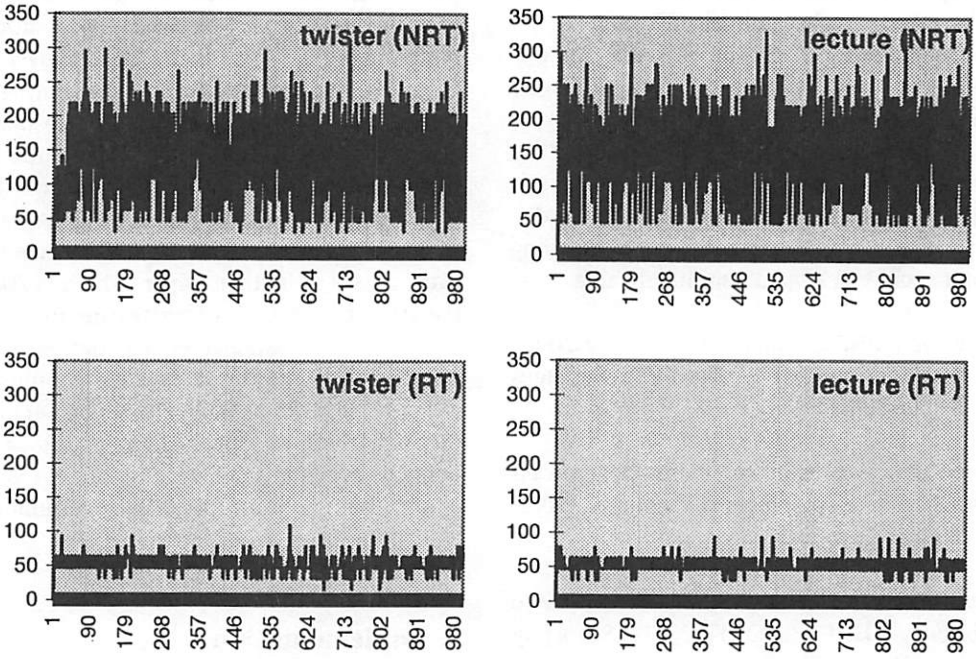


Figure 2: The inter-frame time for the MPEG player that plays the "twister.mpg" and "lecture.mpg" files at 20 frames per second. The y axis measures the inter-frame time in ms, and x axis shows the frame numbers. The top two graphs show the result for the Windows NT kernel scheduler, and the bottom two graphs show the result for our scheduling server with processor reservation for the "twister.mpg" at (100%, 50ms) and "lecture.mpg" at (80%, 50ms).

The graphs in Figure 2 show the measurement of inter-frame time on the MPEG player under the above specified load. The top two graphs show the result for the "twister.mpg" (upper left) and "lecture.mpg" (upper right) under the Windows NT scheduler without our scheduling server. The bottom two graphs show the result for the 20 FPS "twister.mpg" (lower left) with 100% processor reserve every 50 ms, and the 20 FPS "lecture.mpg" (lower right) with 80% processor reserve every 50ms. The "twister.mpg" playback requires more processor time because it contains lots of action with rapid moving background, whereas the "lecture.mpg" playback contains slow moving action with almost no changes in background. Using the Windows NT scheduler, jitter over 200ms (equivalent to 4 frames time) occurs frequently for both MPEG video. Using our scheduling server, jitter over 200ms does not occur at all.

5. Conclusion

Our experiments validate the design and implementation of the soft real-time scheduling server for continuous media processing. Our contribution is that under the control of our scheduling server within the Windows NT environment, (1) RT processes obtain a desired amount of CPU time to satisfy soft real-time requirements; (2) RT processes are monitored and protected against overruns of other RT processes; (3) RT processes have access to timing QoS guarantees and systems services with normal user privileges; and (4) TS processes get a minimum amount of CPU time and do not starve.

References

1. Richard Black, Paul Barham, Austin Donnelly, Neil Stratford. Protocol Implementation in a Vertically Structured Operating System. IEEE LCN, Nov. 1997.

2. Z. Deng, J.W.-S. Liu, J. Sun. Dynamic Scheduling of Hard Real-Time Applications in Open System Environment. *Technical Report* No. UIUCDCS-R-96-1981, Department of Computer Science, University of Illinois at Urbana-Champaign, Oct. 1996.

3. R. Gopalakrishnan. Efficient Quality of Service Support Within Endsystems for High Speed Multimedia Networking. PhD Thesis, Washington University. Dec. 96.

4. Pawan Goyal, Xingang Guo, Harrick Vin. " A Hierarchical CPU Scheduler for Multimedia Operating System". *The proceedings of Second Usenix Symposium on Operating System Design and Implementation*, Seattle WA, Oct 1996.

5. Michael B. Jones, Daniela Rosu, Marcel-Catalin Rosu. "CPU Reservations and Time Constraints: Efficient, predictable Scheduling of Independent Activities". *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, St. Malo, France, Oct. 1997.

6. Chen Lee, Ragunathan Rajkumar, Cliff Mercer. Experience with Processor Reservation and Dynamic QoS in Real-Time Mach. *Multimedia Japan*, 1996.

7. Jun Kamada, Masanobu Yuhara, Etsuo Ono. "User-level Realtime Scheduler Exploiting Kernel-level Fixed Priority Scheduler". *Multimedia Japan*, March 1996.

8. Clifford W. Mercer, Stefan Savage, Hideyuki Tokuda. Processor Capacity Reserves: Operating System Support for Multimedia Applications". *IEEE International Conference on Multimedia Computing and Systems*. May 1994.

9. Klara Nahrstedt, Hao-hua Chu, Srinivas Narayan. QoS-Aware Resource Management for Distributed Multimedia Application. *Technical Report* No. UIUCDCS-R-97-2030, Department of Computer Science, University of Illinois at Urbana-Champaign, Oct. 1996.

10. Jason Nieh, Monica Lam. The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications. *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, St. Malo, France, Oct. 1997.

11. David K.Y. Yau and Simon S. Lam. Adaptive Rate-Controlled Scheduling for Multimedia Applications. *ACM Multimedia Conference*, Boston, MA, Nov. 1996.

12. R. Nagarajan and C. Vogt. Guaranteed-Performance Transport of Multimedia Traffic over the Token Ring. *Technical Report 43.9201*, IBM European Networking Center, IBM Heidelberg, Germany, 1992.

Vassal: Loadable Scheduler Support for Multi-Policy Scheduling

George M. Candea[*]

M.I.T. Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

candea@mit.edu
http://pdos.lcs.mit.edu/~candea/

Michael B. Jones

Microsoft Research, Microsoft Corporation
One Microsoft Way, Building 9s/1
Redmond, WA 98052

mbj@microsoft.com
http://research.microsoft.com/~mbj/

Abstract

This paper presents Vassal, a system that enables applications to dynamically load and unload CPU scheduling policies into the operating system kernel, allowing multiple policies to be in effect simultaneously. With Vassal, applications can utilize scheduling algorithms tailored to their specific needs and general-purpose operating systems can support a wide variety of special-purpose scheduling policies without implementing each of them as a permanent feature of the operating system. We implemented Vassal in the Windows NT 4.0 kernel.

Loaded schedulers coexist with the standard Windows NT scheduler, allowing most applications to continue being scheduled as before, even while specialized scheduling is employed for applications that request it. A loaded scheduler can dynamically choose to schedule threads in its class, or can delegate their scheduling to the native scheduler, exercising as much or as little control as needed. Thus, loaded schedulers can provide scheduling facilities and behaviors not otherwise available. Our initial prototype implementation of Vassal supports two concurrent scheduling policies: a single loaded scheduler and the native scheduler. The changes we made to Windows NT were minimal and they have essentially no impact on system behavior when loadable schedulers are not in use. Furthermore, loaded schedulers operate with essentially the same efficiency as the default scheduler. An added benefit of loadable schedulers is that they enable rapid prototyping of new scheduling algorithms by often removing the time-consuming reboot step from the traditional edit/compile/reboot/debug cycle.

In addition to the Vassal infrastructure, we also describe a "proof of concept" loadable real-time scheduler and performance results.

1. Introduction

A primary function of operating systems is to multiplex physical resources such as CPU, memory, and I/O devices among application programs. The CPU is one of the primary resources, and hence, it is important to schedule it effectively. This raises the question: what is the best algorithm to schedule tasks on the available CPUs?

The answer, of course, strongly depends upon the mix of tasks to be run, the demands that they place on the different resources in the system, and the relative values of the various outcomes that will result from different scheduling decisions. In the limit, for an operating system to perform optimal scheduling of its CPUs, it would need perfect knowledge of the future behavior and requirements of all its applications.

Most "general purpose" systems have used algorithms which know either nothing or next-to-nothing about the actual CPU resource needs of the tasks being scheduled. Examples include "First-Come, First Served" used in early batch systems and Round-Robin in multi-tasking systems. Later algorithms such as Priority Queues ([Corbató & Daggett 62], [Lampson 68]), Fair Share Scheduling ([Kay & Lauder 88]), and typical dynamic priority boost/decay algorithms still had the property that they were essentially ignorant of the actual CPU needs of their applications.

Imperfect, but nonetheless adequate, future knowledge is possible for some fixed task sets with well-characterized computation patterns. Whole families of scheduling disciplines have arisen in the computer systems research community to provide appropriate scheduling for some such classes. Examples are: Earliest Deadline First ([Liu & Layland 73]), Weighted Fair Queuing ([Clark et al. 92]), Pinwheel Scheduling ([Hsue & Lin 96]), and Proportional Share CPU allocation mechanisms ([Waldspurger 95], [Goyal et al. 96]), plus techniques such as Rate Monotonic Analysis ([Liu & Layland 73]) and Priority Inheritance ([Sha et al. 90]). Similarly, Gang Scheduling ([Ousterhout 82]) and Implicit Coscheduling ([Dusseau et al. 96]) were developed for parallel workloads where the forward progress of members of a task set is closely dependent upon the progress of other tasks in the set.

But today's general purpose operating systems do not provide such specialized scheduling algorithms. Some of the more popular operating systems provide a primitive differentiation between the different scheduling classes by mapping them onto different priorities (e.g., System V Release 4 [Goodheart & Cox 94], Windows NT [Solomon 98]) and then scheduling higher priority tasks more often or for longer periods of time. However, it is extremely

[*] The research described in this paper was done while George Candea was a summer intern at Microsoft Research.

hard to properly map requirements such as predictability, throughput, fairness, turnaround time, waiting time, or response time onto a fixed set of priorities. Moreover, different applications may use different mappings, defeating their purpose. For instance, when co-existing applications do not share coordinated priority mappings or goals, it is common for applications wanting "real-time performance" to raise their priority to the highest one available under the assumption that they are "the most important" task in the system — a phenomenon known as "priority inflation". Priorities are, at best, a rather primitive way of describing the relative performance requirements of the threads and processes that belong to the different classes.

Other systems ([Northcutt 88], [Jones et al. 97], [Nieh & Lam 97], etc.) have tried to strike a compromise, providing some application timing and resource advice to the system, giving it imperfect, but useful information upon which to base scheduling decisions.

Such large numbers of different scheduling algorithms are an indication that scheduling is, and will likely remain, an active area of research. No one algorithm will work best in all cases (despite many valiant attempts by system builders to demonstrate otherwise). In fact, [Kleinrock 74] shows that any scheduling algorithm that favors a certain class of tasks will necessarily hurt another class. A single scheduling policy will always represent a compromise and the service offered by the system will unavoidably reflect this compromise.

Our Proposed Solution

As discussed above, we believe that any particular choice of scheduling algorithm will fail to address the needs of some classes of applications, particularly when independent applications with different scheduling requirements are concurrently executed. Rather than attempting to devise yet one more "good compromise" we explored a different approach.

We decided to find out whether we could dynamically extend systems with scheduling algorithms. While nearly all modern established operating systems can be partially extended via loadable modules (e.g., Linux, Solaris, Windows NT) and extensible systems are a very active area of research ([Bershad et al. 95], [Engler et al. 95], [Seltzer & Small 97]), none of these systems allowed arbitrary scheduling policies to be implemented as extensions — motivating our work on Vassal.

The results were quite positive: it was straightforward to modify a modern commercial operating system, in this case Windows NT 4.0, in order to allow independently developed and compiled schedulers to be dynamically loaded into (and unloaded from) the operating system at run-time.

The loaded schedulers can take control of as many or as few of the system's scheduling decisions as desired. For instance, in our implementation, the existing Windows NT scheduler was retained, so a loaded scheduler can always fall back upon the default system scheduler if it chooses not to make a particular scheduling decision. And in the case when no loadable scheduler is present, the system works exactly as it would have were the loadable scheduler support not there.

The modifications resulted in no measurable performance penalty when loadable schedulers are not in use. Furthermore, loadable schedulers can operate with nearly the same efficiency as the native system scheduler. Finally, having a loadable scheduler infrastructure makes it easy to experiment with different schedulers, providing special-purpose scheduling on a general-purpose system.

The present Vassal implementation is clearly a prototype, with some limitations. For instance, at present we only support the simultaneous coexistence of two schedulers: the Windows NT scheduler and a single loaded scheduler. Nonetheless, we believe that the techniques and results obtained with the prototype will remain valid once these limitations are removed. For more on this topic, see Section 8.

In the following sections we provide some background on the system we started with, describe the particular system we built in more detail, and then show what transformations we made to the vanilla system. We present a "proof-of-concept" real-time scheduler that we wrote, followed by performance measurements. We then discuss the experiences we had while building and experimenting with the loadable multi-policy scheduler support and conclude.

2. Background

This section provides background information on some of the features of Windows NT 4.0 relevant to our loadable scheduler work. We describe the native scheduler and its implementation, and also present briefly the NT driver model.

Windows NT Scheduling Model

Windows NT uses threads as its basic schedulable unit. Threads can exist in a number of states, the most important ones being: *Running* (executing on a processor), *Standby* (has been selected for execution on a processor and is waiting for a context switch to occur), *Ready* (ready to execute but not running or standing by), *Waiting* (either waiting on a synchronization object, such as a semaphore, waiting for I/O to complete, or has been suspended), and *Terminated* (the thread is not executing anymore and may be freed or recycled).

The thread dispatcher is responsible for scheduling the threads and it does this based on two thread characteristics:

- priority (higher priority threads are scheduled before lower-priority ones);
- processor affinity (threads may have preferences for a certain processor in multi-processor systems and this is accounted for when scheduling it).

Windows NT provides a set of 32 priorities, which are partitioned into three groups:

1. The *Real-Time* scheduling class, which includes the highest priorities in the system (16-31). Threads belonging to this class can gain exclusive use of all scheduled time on a processor if there is no runnable thread with a higher priority in the system.
2. The *Variable* priority scheduling class, which includes priorities 1-15. Threads belonging to this class are subject to priority drops (e.g., when the thread's time quantum runs out) or priority boosts (e.g., when awaited I/O completes). As can be seen, the priority of a CPU-bound thread in this class decays over time, unlike threads in the *Real-Time* class.
3. Priority 0 is the lowest priority and is reserved for the so-called *idle* thread. This thread runs whenever there is no ready thread available in the system.

It is important to note that under Windows NT, not all CPU time is controlled by the scheduler. Of course, time spent in interrupt handling is unscheduled, although the system is designed to minimize hardware interrupt latencies by doing as little work as possible at interrupt level and quickly returning from the interrupt. The mechanism that ensures this is Deferred Procedure Calls (DPCs). DPCs are routines executed within the Windows NT kernel in the context of no particular thread in response to queued requests for their execution. For example, DPCs check the timer queues for expired timers and process the completion of I/O requests. The way hardware interrupt latency is reduced is by having interrupt handlers queue DPCs to finish the processing associated with the interrupt and then return. Due to their importance, DPCs are executed whenever a scheduling event is triggered, prior to starting the scheduled thread, and they do not count against any thread's time slice.

Windows NT Scheduler Implementation

A scheduling request can be generated by a number of events. Some of these are:

- The time quantum of a running thread expires.
- Thread state changes, such as when a thread enters the *Ready* state or when the currently running thread enters the *Waiting* or *Terminated* state.
- When the priority or affinity of a thread in the system is changed from outside the scheduler (e.g., by the *SetThreadPriority()* call).

Whenever the hardware clock generates an interrupt, the Hardware Abstraction Layer (HAL), which exports a virtual machine to the NT kernel, processes the interrupt and performs platform-specific functions. After that, control is given to the kernel. At this point the kernel updates a number of counters, such as the system time, and inspects the queue that contains timers. For every expired timer it queues an associated DPC. After that it decrements the running thread's time quantum and checks whether it has run out. If yes, it issues a *DISPATCH* software interrupt on the corresponding processor. All events that trigger scheduling raise *DISPATCH* software interrupts.

The *DISPATCH* software interrupt then invokes a kernel handler which first runs all the queued DPCs. After this, the thread dispatcher is ready to make a scheduling decision.

The set of data structures used by the NT dispatcher are collectively known as the dispatcher database. This set contains information about which threads are running on which processors, which threads are ready to run, etc. The most important data structure is the set of thread queues that keep track of threads in *Ready* state; there is one such queue for each priority (except 0). Whenever scheduling is triggered, the scheduler/dispatcher walks the *Ready* thread queues in decreasing order of priority. It then schedules the first thread it finds, provided the thread's processor affinity allows it to be scheduled on the free processor. The thread is prepared for execution (if not currently running), a context switch is performed, and then the DISPATCH service routine returns from the interrupt.

The NT kernel provides a system call, *NtSetTimerResolution()*, which allows the frequency of clock interrupts to be adjusted. Specifically, when an application needs high resolution timers, it may choose to lower the time between clock interrupts from the default (typically 10ms) to the minimum supported (typically 1ms).

Of particular importance to scheduling is the fact that the HAL does not export a programmable timer to the kernel, which denies the kernel the ability to reschedule at a precise point in time. For instance the programmable timer available on x86 PCs is used by the HAL as a countdown timer that is repeatedly set to the current interval between interrupts (so it is essentially used as a periodic timer). Most other non-real-time operating systems running on the x86 do the same.

Windows NT Driver Model

Drivers in Windows NT do much more than traditional device drivers, which just enable the kernel to interface to hardware devices. NT drivers are more of a general mechanism by which NT can be extended. For example, under NT, filesystems, network protocol implementations, and hardware device management code are all separately compiled, dynamically loadable device drivers. Drivers reside in kernel space, can be layered on top of each other, and communicate among themselves using I/O Request Packets (IRPs) in a manner reminiscent of UNIX System V Streams modules [Ritchie 84]. Applications typically send and receive data to and from the drivers via the same path.

3. Loadable Scheduler Design

This section describes the design of the infrastructure that allows scheduling policies to be loaded and unloaded at run-time. It is intended to provide a guide to implementing such a system, while omitting OS-specific details, which are discussed in the following section.

The basic idea is to modify the thread dispatcher inside the kernel so that it handles multiple scheduling policies. It is the *decision making* component of a scheduler that contains all the policy, so we chose to externalize the decision-making by encapsulating it in loadable drivers, while leaving all the dispatching mechanism in the kernel. We wish to replace the statement "the dispatcher decides which thread to run" with "the dispatcher queries the schedulers for which thread to run." The maintenance of the thread queues, being a chore specific to the decision making process, is done by the external schedulers themselves. The in-kernel dispatcher simply expects a reference to the appropriate thread data structure from the scheduler it queries. Figure 1 shows the conceptual architecture of Vassal with an example in which there are four tasks running on the system (*T1, T2, T3, T4*) and there are currently two schedulers available (*A* and *B*).

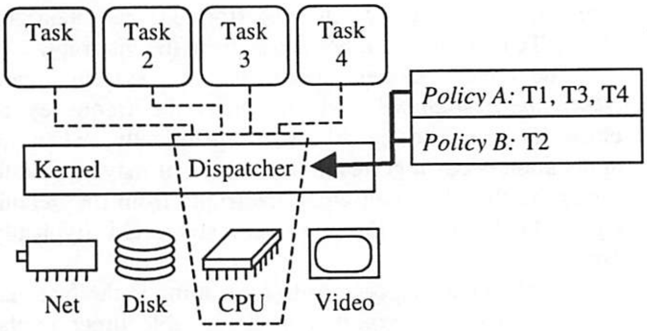


Figure 1: The Loadable Scheduler Infrastructure

The Vassal dispatcher manages the schedulers and dispatches/multiplexes messages between the kernel and the schedulers. It is responsible for:

- Receiving scheduling requests from the kernel and deciding which scheduler to query.
- Relaying the scheduler's response to the application.
- Deciding whether a scheduler that is attempting to load would conflict with already existing schedulers. The scheduler is loaded only if there are no conflicts.
- Enabling communication between threads and schedulers.

Scheduling

When a *DISPATCH* software interrupt is generated, an interrupt service routine is invoked and eventually hands control over to the dispatcher. The dispatcher then decides which scheduler to query. In our current model, we simply use a hierarchy of schedulers (with the external scheduler being at the top), so that if a higher level scheduler does not have a ready thread, then the next one (in descending order) is queried. Section 8 describes other possible ways of managing relationships between schedulers. Once a scheduler responds with a runnable thread, the dispatcher can perform a context switch (if necessary), schedule the thread, and return.

It may seem that, by using a hierarchy of schedulers, we are essentially making the decisions based on a set of priorities. However, there is a significant difference between scheduler hierarchy and thread hierarchy: the scheduler priorities have nothing to do with what the threads think is more important (which would motivate their choices for priorities) rather it has to do with implicit relationships between the schedulers that result from their CPU resource requirements.

Thread Creation

Newly created threads initially execute using the system's default scheduler. It can then make explicit requests to be scheduled by other schedulers. Other approaches could have equally well been taken. For instance, a thread could inherit its parent's scheduling class, or an explicit scheduler parameter could have been provided to an extended thread creation call.

Here is an example scenario. A task (e.g., *T2*) is created and associated with the default scheduler (e.g., *A*). After running for a while, *T2* makes a system call to inform the system that it now wants to switch to another scheduler. The kernel relays this information to the desired scheduler, which in turn removes the task from the previous scheduler's jurisdiction. From this point on, the new scheduler has sole ownership over *T2*'s schedule (until the task decides to switch again).

Communication between Threads and Schedulers

For optimal scheduling decisions, every scheduler needs semantic information about the intentions and requirements of the threads under its jurisdiction. Once provided with this information, schedulers can make the appropriate scheduling decisions. For this reason, the operating system interface needs a system call that allows threads to communicate with a scheduler of their choice. The dispatcher receives this stream of messages from the kernel and demultiplexes it. By this means, a thread could inform its scheduler that it wants to communicate with a thread on another processor and, thus, the scheduler should attempt to schedule that thread concurrently with the requesting thread.

One question that arises naturally is whether the use of the external schedulers would negatively impact performance, given that on every timer interrupt there would be a query going out to these schedulers. However, the critical path followed by these queries turns out to be very short, because the only added time is that of performing a small number of memory reads from non-pageable memory. Remember that schedulers (being drivers) reside in the kernel address space. As our results suggest, this added overhead is negligible and is clearly offset by the gains in scheduling performance. Also, in a multi-threaded kernel such as NT's, it is possible for the schedulers to avoid making decisions on the critical path by having their decisions ready before they are queried.

Another issue we considered was whether a thread that was not selected for execution by its current scheduler could be selected by another scheduler. For instance, one

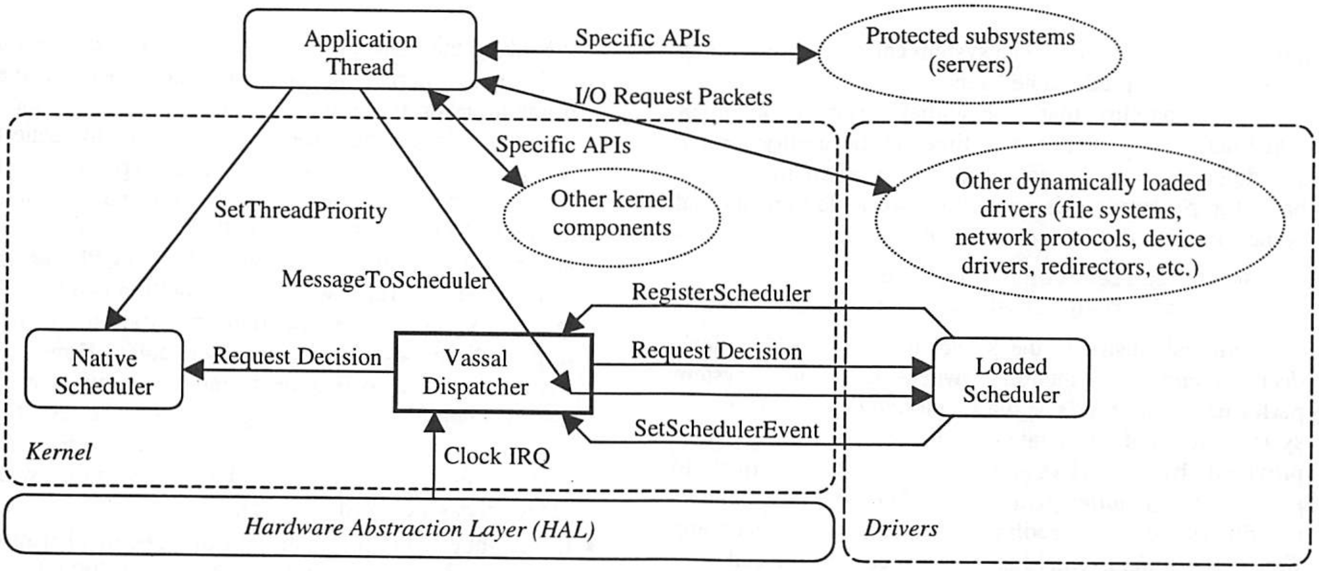


Figure 2: Integration of Vassal into Windows NT 4.0

could view this as happening in [Jones et al. 97] when a thread's CPU reservations and/or time constraints do not cause it to be selected, but it is nonetheless selected by the default round-robin policy. As this is more general, we opted to allow this (while also making it possible for a scheduler to prevent it). One example of our use of multi-policy scheduling is that a thread not selected by the sample scheduler described in Section 5 can typically still be selected by the default scheduler.

4. Vassal Implementation

This section describes the Vassal implementation, including the modifications and additions we made to the Windows NT kernel in order to support multiple schedulers. An overview of the structure of the Vassal implementation within Windows NT is shown in Figure 2.

First of all, we needed to add a way for the scheduler/driver to notify the system that it is being loaded. For this we added a kernel function for scheduler drivers:

```
RegisterScheduler(scheduler,
    decision_maker, message_dispatcher)
```

The *decision_maker* parameter to *RegisterScheduler()* is the address of a function to be called when a scheduling decision needs to be made. This function must either return a runnable thread to be scheduled or NULL, which indicates that the loaded scheduler has no opinion as to which thread should be scheduled. In this case, Vassal calls the next scheduler in the hierarchy, allowing it to make the scheduling decision. (In the present prototype this means the decision is always delegated to the built-in scheduler.)

The *message_dispatcher* parameter to *RegisterScheduler()* is the address of a function to be called to handle messages from threads to the scheduler. This routine handles requests sent via the *MessageToScheduler()* system call, which is described below.

In response to the *RegisterScheduler()* call, Vassal saves these parameters of the loaded scheduler and activates the new scheduler. We also provide a matching *UnregisterScheduler()* function, which allows a scheduler to be unloaded cleanly (using the customary procedure for unloading drivers).

For applications to be able to communicate with the schedulers, we added a new system call:

```
MessageToScheduler(scheduler, buffer,
    buflen)
```

by which an application can send a message (in *buffer*) to a loaded scheduler. Upon receiving this call, the scheduler's *message_dispatcher* routine will be invoked, passing the buffer contents as a parameter. The scheduler then performs the action specified by the buffer contents. When the routine completes, its return value is returned to the application. Note that *MessageToScheduler()* need not return immediately; it is free to block or take any action that a regular driver could take.

Of course, given that loaded schedulers are full-fledged Windows NT device drivers, one might ask why we added the *MessageToScheduler()* system call at all — why not just use standard device *Read()* and *Write()* operations to communicate with the scheduler? As described in Section 2, these user-space operations generate I/O Request Packets (IRPs) that would travel through the I/O subsystem and eventually reach the scheduler/driver. In fact, we initially did use IRPs to communicate with loaded schedulers.

We added the *MessageToScheduler()* system call because we found that, in practice, the amount of time it would take requests to reach the scheduler via the standard I/O path was too unpredictable for the time-critical services that the external scheduler is intended to provide, especially for the kinds of real-time schedulers we were attempting to construct. The system call gave Vassal both lower and more predictable latency.

One capability that is essential for many kinds of schedulers (in particular, real-time) is the ability for the scheduler to cause an action at a designated time. As a basis for providing this capability, we added an internal kernel function available to schedulers:

```
SetSchedulerEvent(scheduler,
    performance_counter_reading)
```

This call instructs the kernel to call the scheduler's *decision_maker* function whenever the system performance counter's value (a monotonically increasing system-provided low-latency 64-bit real-time timer provided by the HAL) is greater than or equal to *performance_counter_reading*. This facility allows schedulers to set deadlines. There is a matching *CancelSchedulerEvent()* function that cancels the call.

To support the functionality described above, we made modifications to the routine that services the *DISPATCH* software interrupts. Its function is to process the Deferred Procedure Call (DPC) list, query the scheduler if necessary and then perform a context switch if a new thread has been selected for execution on the processor. We needed to add a hook that would substitute calling the loaded scheduler's decision function, when one is available, in place of querying the built-in scheduler. Additionally, for the servicing of scheduler events, we needed to further modify this routine so that it would check on every clock interrupt whether the performance counter reading had reached the desired value and, if so, trigger a scheduling decision.

Given that drivers do not have full access to kernel data structures, we also needed to add a number of simple methods that allow schedulers to manipulate and gain access to parts of those data structures. This new functionality includes finding which CPU is currently being scheduled, what the status of a thread is, removing/adding threads from/to the pool of natively scheduled threads, and preempting a thread.

5. A Sample Scheduler

We wrote a simple real-time scheduler as a proof of concept. It was rather straightforward (116 lines of C code). This scheduler allows threads to request that they be scheduled at a particular time, which exercises the key operation needed to implement more interesting time-based scheduling primitives, such as time constraints [Jones et al. 97]. Using this scheduler, we can easily write timers that have a much higher accuracy and resolution than the multimedia timers offered by Windows NT (multimedia developers choose to use buffering and a number of other tricks to circumvent the limitations of NT timers; with our scheduler, this would be unnecessary). Section 6 shows the measured latencies for these timers.

In order for the sample real-time scheduler to achieve its goal, we made two important decisions:

- We added the concept of a *settable event* and added a call to the kernel interface (as described in the previous section) that would allow a driver to set such an event. In essence, we enable a driver to request that it be given control of the CPU at a specific point in time based on the value of the performance counter. This counter is typically both a precise and accurate way of measuring time. On x86 CPUs using standard HALs, for instance, its resolution is 0.838μs. It might seem that we essentially modified the kernel to provide support for a specific scheduler. However, we made this modification because we saw it as a useful feature for many types of schedulers (for instance if they need to perform certain actions at regular intervals or they need to synchronize with other processes based on time).
- If the requested time constraint cannot be met because it is very short (e.g., a time constraint of 200μs), we choose to spin in a loop until the time comes to schedule the thread. We cannot presently count on a higher accuracy than 1ms from the HAL (see Section 8), so we used this admitted hack in the sample scheduler (not the kernel code) to achieve higher resolution for one thread.

Note that we have not implemented on-demand loading of schedulers but it would be very simple to do. Currently, in order to load the real-time scheduler and have it coexist with the native scheduler the system administrator uses the Control Panel and select the appropriate driver. It can be unloaded in the same way.

The following code snippet shows a simple thread using the real-time scheduler.

```
/* Tell system to use the real-time scheduler */
status = MessageToScheduler (rt_sched, {JOIN});
if (status != SUCCESS) {
    error (“Could not join R/T scheduling class.”);
}
/* Calculate how long our loop iterations take */
estimate = Calibrate();
/* Start the loop 1 ms from now */
status = MessageToScheduler (rt_sched, {SET, 1000});
if (status != SUCCESS) {
    error (“Could not set deadline.”);
}
/* We want one iteration every 300 μs */
while (1) {
    status = MessageToScheduler (rt_sched, {SET, 300 –
        estimate});
    ...
}
```

The function *Calibrate()* computes an estimate of how long it will take to perform each loop iteration. Notice the use of a single system call to communicate with the scheduler. Figure 3 details the actions that are triggered by the various steps in the program.

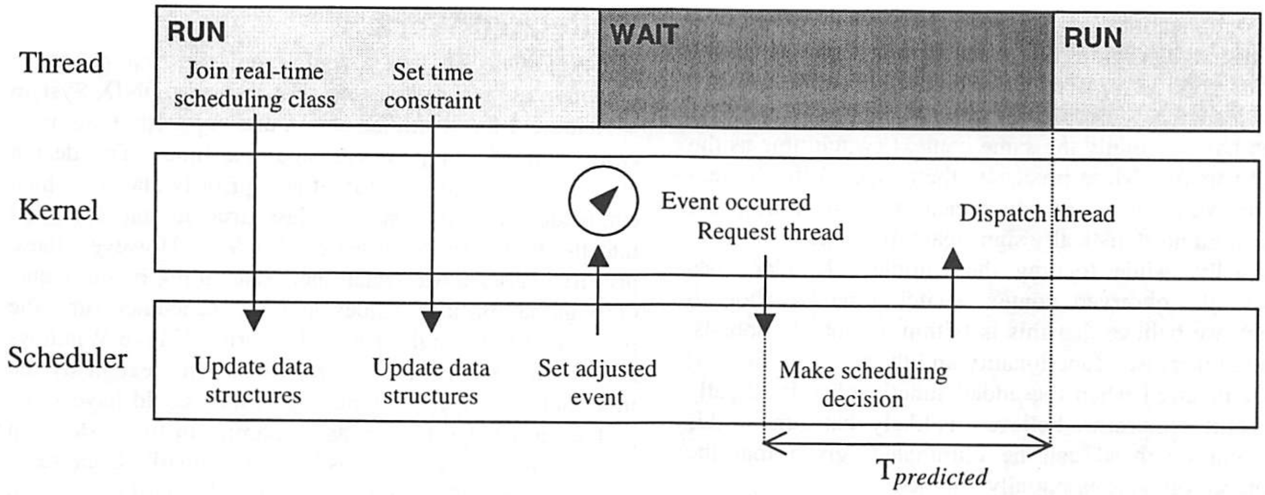


Figure 3: The actions taken for the execution of the first part of the sample code

The first two system calls translate into messages delivered directly to the loaded scheduler; if the thread's request can be satisfied, it returns a status of SUCCESS. The scheduler then updates its data structures to reflect the thread's requirements and sets the event mentioned above to the appropriate performance counter reading (the constraint interval of 1ms is large enough to do this). Then, when the event occurs, the scheduler is notified and asked for a runnable thread. As a result, the scheduler sees that the deadline has arrived for the requesting thread and instructs the kernel to schedule it. Note that the setting of the event takes into account the fixed time $T_{predicted}$, which is the platform-dependent time it takes for a message to make it through the critical path (as described in Section 3).

An important property of the scheduler is that if it is unable to satisfy the thread's request, it informs it *right away* (via the return code of *MessageToScheduler*). This is in contrast to what happens on most general purpose operating systems, where the thread expecting to meet a certain deadline finds out it cannot make it barely after it has already missed the deadline. Using the information given by our real-time scheduler, the application could adjust and decide to take some action that can compensate for the missed deadline.

The current method of keeping a thread spinning if the time constraint is very small is not a very clean solution. However, other means of obtaining the desired accuracy would require more substantial changes to the underlying kernel and, more importantly, to the HALs. Such solutions would be more difficult to adopt.

6. Results

Code Size Results

The Vassal changes made to the Windows NT kernel to support multi-policy scheduling added 188 lines of C code, added 61 assembly instructions, and replaced 6 assembly instructions.

The proof-of-concept external scheduler described earlier required only 116 lines of C code and no assembly language. We believe these are extremely low code size numbers for the increased functionality that we achieved.

Performance Results

One of the primary criteria against which any loadable scheduler support would be judged when added to a production operating system would be whether it makes things any worse for applications not using it. We are happy to report that, when a loadable scheduler is not in use, our changes have no performance impact on system performance.

One value that the use of loaded schedulers might be expected to change is the context switch time. To measure the performance impact of our changes, we ran a program that recorded actual context switch times as observed from user space by 10 threads, over a period of 10 seconds. Times were collected using the Pentium cycle counter on a 133MHz Pentium PC. Table 1 shows the results.

System Version	Median	Avg.	Std. Dev.
Vanilla NT 4.0 (released)	17.03	18.71	4.17
Vanilla NT 4.0 (rebuilt)	19.95	19.88	1.64
Vassal (no loaded scheduler)	19.71	19.71	1.56
Vassal (sample scheduler loaded)	21.32	21.17	1.28

Table 1: Measured context switch times on a Pentium-133 running the original and the modified systems (in µs).

We first explain the difference in the first two sets of data. The "Vanilla NT 4.0 (released)" figures are from the product version of NT 4.0 Workstation. The "Vanilla NT 4.0 (rebuilt)" figures are for a kernel built from the identical NT 4.0 sources with no modifications. However, the rebuilt version does not contain all of the binary optimizations contained in the product version. This explains why it is roughly 6-7 percent slower than the product version. All Vassal versions are built with the same optimizations as the rebuilt version. Thus, the rebuilt version provides the correct basis for comparison.

The good news is that the Vassal version of NT 4.0 (with loadable scheduler support) with no scheduler loaded has essentially the same context switch time as the rebuilt version. More precisely, their times differ by less than the variations seen while measuring the times, and thus exhibit no statistically significant difference.

Finally, while loading the sample scheduler does increase the observed context switch time by about 8 percent, we believe that this is within acceptable bounds, given the increased functionality and the fact that this cost is only incurred when the added functionality is actually used. Furthermore, we believe it is likely that some of this 8 percent overhead can be eliminated, given that the current prototype is essentially untuned.

Sample Scheduler Results

The proof-of-concept sample real-time scheduler implements one primitive thread scheduling operation not otherwise found in standard Windows NT: a precisely timed thread wakeup. In a loop, this can be used to perform periodic processing, such as doing a short operation once every millisecond.

Standard Windows NT contains periodic multimedia timers that are designed for this kind of periodic processing. This section compares the effectiveness of doing periodic wakeups once per millisecond with the loaded sample scheduler and NT's multimedia timers. Table 2 shows the results.

Method	Min.	Max.	Avg.	Std. Dev.
NT Multimedia Timers	75	1566	996	82
Sample Scheduler Events	996	1485	1002	21

Table 2: Periodic wakeup times on a Pentium-133 using multimedia timers on the original system and the sample scheduler's events on the modified system (in µs). The desired value is 1ms.

Two differences are evident in the results. First, while using multimedia timers some wakeups occurred extremely early, as much as 925µs too soon. With the sample scheduler wakeups occurred at most 4µs early.

Second, predictability of the wakeups with the sample scheduler is significantly better than with the multimedia timers. The standard deviation of the sample scheduler data is only a quarter of that for the multimedia timers.

With both methods, some samples occurred up to ~0.5ms late. Initial studies indicate that these samples are due to interrupts, DPCs, and other non-scheduled system activities, although this bears further investigation.

While extremely simple, we believe that this example begins to show the potential of extending the scheduling policies available to applications through the use of loadable schedulers.

7. Related Work

Windows NT was certainly not the first operating system to use priority classes. For instance, UNIX System V Release 4 has a similar notion and supports three basic types (time-sharing, system, and real-time). The design allows for the incorporation of new priority classes, which can be described by a special class structure and compiled into the kernel ([Goodheart & Cox 94]). However, these priority classes always map their scheduling requirements onto global priority values and the scheduler runs the process with the highest global priority. Unlike Windows NT, these priorities are controlled to some extent by the in-kernel class-specific functions, which could have some global knowledge of the tasks running in the system. In spite of this, such a system is limited, primarily because:

- New scheduling policies need to be hard-coded into the kernel; this implies the need for source code, which may not always be available. In addition, a programmer writing a new scheduling policy may not want to have to dive into kernel internals in order to implement the policy. If the programmer writing extensions for a system needs to know as much about the system as the person who wrote it, then such extensions will likely never be written.
- The first disadvantage implies the second: scheduling policies cannot be dynamically added or removed at runtime, which makes the system less flexible and makes the debug cycle longer.

Solaris does allow scheduling classes to be dynamically loaded into the kernel, although these classes are still subject to the restriction that they map their scheduling decisions onto a global thread priority space.

A number of recent efforts are aimed at making operating systems extensible [Bershad et al. 95], [Kaashoek et al. 97], [Seltzer & Small 97]. However these do not have the same goals as Vassal and do not provide the same facilities.

The one that offers scheduling features closest to Vassal is SPIN [Bershad et al. 95]. It offers applications the ability to provide their own thread package and scheduler, which can then execute in kernel space. This way applications can define their own thread semantics. A global scheduler implements the primary scheduling policy, which is a priority scheduler, with round-robin execution within each priority. The global scheduler is not extensible. Application-defined schedulers are layered on top of the global scheduler. However, this global scheduler may reclaim the CPU from any given strand, therefore no application-defined scheduler has any guarantee of when it will receive time to schedule. Note that SPIN is addressing a different problem domain than we are. It loads Modula-3 extensions into the kernel, which are limited by the type-safe characteristics of the language. The purpose of this is to achieve protection. In our model, the "extensions" (i.e., the drivers) are trusted and therefore we do not need to protect against them.

Another body of work related to ours pertains to support for hierarchical scheduling. In such systems, time allocated to one scheduler is sub-allocated to other dependent schedulers. [Ford & Susarla 96] describes one such system.

Finally, [Deng & Liu 97] propose a new hierarchical scheduler for Windows NT that provides timing guarantees for multiple independent hard real-time applications, while continuing to run normal applications.

8. Limitations and Future Work

The present Vassal implementation is definitely a prototype, and is subject to certain restrictions and limitations. This section describes some of the present implementation limitations and possible future work.

Some of the most severe limitations that Vassal loadable schedulers are presently subjected to are actually limitations of the underlying Windows NT system upon which they were built. In particular, loadable schedulers (as well as the built-in scheduler) cannot exercise complete control over what code is run when.

As described in Section 2, all system timer services are actually provided to the kernel through the Hardware Abstraction Layer (HAL), which does not provide an interface to cause an interrupt at a precise time. Timer interrupts are only generated at the clock tick frequency, which while settable, is not settable to resolutions finer than 1ms for the standard x86 ISA PC HAL. Thus, sub-millisecond preemptive scheduling is effectively not implementable under NT in a clean way. The PC hardware does support this capability; removing this restriction would involve adding precise non-periodic interrupt capabilities to the HAL interface and HAL implementations.

Other sources of unscheduled time include DPCs and interrupts. The system could be modified to schedule DPCs, although this would result in both additional overhead and greater code complexity. We suspect that very little can be done to improve interrupt latencies beyond what the production system has already achieved.

Another limitation present in the current Vassal implementation is that only one external scheduler may be loaded at once. This is partly due to the slightly higher implementation complexity involved in managing the state of multiple loaded schedulers but actually mostly due to more fundamental policy issues that would need to be resolved to support arbitrary numbers of coexisting schedulers in a general way.

The key policy issue is this: schedulers might have conflicting goals, or equivalently, might be running applications with conflicting goals. For instance, if two schedulers want to run different code at precisely the same time on the same processor, the schedulers are in conflict. The question is what to do when schedulers' goals collide.

Our present implementation solves this by simple fiat — the loaded scheduler always takes precedence over the built-in scheduler. Provided the built-in scheduler is occasionally given an opportunity to run, this does not violate its invariants, since it does not depend upon being run at any particular time, unlike real-time schedulers.

Indeed, a similar strict decision hierarchy could be used among more than two coexisting schedulers, although this admits the possibility of loading two schedulers (or applications using them) with mutually incompatible requirements.

One way to achieve conflict resolution would be by allowing loadable schedulers to describe the ways they will use the CPU. For example, one scheduler may need "hard time" (i.e. it needs the CPU at definite moments in time) while another one may only require a "share" of the CPU (no matter when it occurs). A simple way to describe such requirements would be by using predefined usage patterns (i.e. "hard time," "proportional share," etc.). A more sophisticated solution would involve the use of a formal language by which schedulers could express their requested usage pattern.

Based on these usage descriptions, the dispatcher could determine whether a scheduler that is attempting to load would conflict with an already existing scheduler. For instance, multiple schedulers with "hard time" requirements will typically not coexist. This way, conflicts could be usefully detected and avoided or prevented, given sufficiently accurate CPU requirements specifications for each scheduler and/or {scheduler, application} pair.

9. Conclusion

This paper has described an infrastructure that allows multiple scheduling policies to coexist simultaneously in an operating system. It has shown that scheduling policy modules may be developed separately from the base operating system, and that these policy modules can be dynamically loaded and unloaded from a running system as needed. It has demonstrated that both this infrastructure and loadable schedulers can be simple to implement, even for commercial operating systems of the complexity of Windows NT.

The addition of these capabilities resulted in no measurable performance penalty when loadable schedulers are not in use. Furthermore, loadable schedulers can operate with nearly the same efficiency as the native system scheduler.

We found that having a loadable scheduler infrastructure makes it easy to experiment with and utilize different schedulers, providing special-purpose scheduling within a general-purpose system. One unanticipated benefit of loadable schedulers is that they enable rapid prototyping of new scheduling algorithms by often removing the time-consuming reboot step from the traditional edit/compile/reboot/debug cycle.

Furthermore, and possibly most importantly, providing this form of operating system extensibility frees the operating system designer from having to anticipate all possible scheduling behaviors required by applications. Instead, it allows new scheduling policies to be developed and used without requiring changes to the base operating system.

While this work was conducted within the Windows NT 4.0 kernel, we believe that many of the ideas and implementation techniques presented in this paper should be applicable to other operating systems, providing them the same scheduling flexibility that is present in Vassal.

Acknowledgements

We want to thank Bill Bolosky, Rich Draves, Johannes Helander, and Rick Rashid for their ideas, suggestions, and practical advice during this project. Andrea Arpaci-Dusseau provided useful information on Solaris scheduling classes. Emin Gün Sirer explained fine points of SPIN scheduling. Thanks are also due to Patricia Jones for her technical editing and moral support and to George Jones for the timely use of his computing environment.

References

[Bershad et al. 95] Bershad, S. Savage, P. Pardyak, E.G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers, "Extensibility, Safety and Performance in the SPIN Operating System," *Proceedings of the 15th ACM Symposium on Operating System Principles*, pp. 267-284, Dec. 1995.

[Clark et al. 92] D. D. Clark, S. Shenker, and L. Zhang, "Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism" *Proceedings of ACM SIGCOMM '92*, pp. 14-26, Aug. 1992

[Corbató & Daggett 62] F.J. Corbató , M. Merwin-Daggett, R.C. Daley, "An Experimental Time-Sharing System", *Proceedings of AFIPS Spring Joint Computer Conference*, pp. 335-344, 1962.

[Deng & Liu 97] Z. Deng and J. W.-S. Liu. "Scheduling Real-Time Applications in an Open Environment", *Proceedings of the 18th IEEE Real-Time Systems Symposium*, San Francisco, pp. 308-319, Dec. 1997.

[Dusseau et al. 96] A.C. Dusseau, R.H. Arpaci, D.E. Culler, "Effective Distributed Scheduling of Parallel Workloads," *Proceedings of Sigmetrics '96 Conference on the Measurement and Modeling of Computer Systems*, pp. 25-36, May 1996.

[Engler et al. 95] D.R. Engler, M.F. Kaashoek, J. O'Toole Jr., "Exokernel: an Operating System Architecture for Application-Specific Resource Management," *Proceedings of 15th ACM Symposium on Operating System Principles*, pp. 251-266, Dec. 1995.

[Ford & Susarla 96] B. Ford and S. Susarla, "CPU Inheritance Scheduling," *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pp. 91-105, Oct. 1996.

[Goodheart & Cox 94] B. Goodheart and J. Cox, *The Magic Garden Explained: the Internals of UNIX System V Release 4, an Open Systems Design*, Sydney, Prentice Hall, 1994.

[Goyal et al. 96] P. Goyal, X. Guo, H. M. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pp. 107-121, Oct. 1996.

[Hsue & Lin 96] C. Hsueh and K. Lin, "Optimal Pinwheel Schedulers Using the Single-Number Reduction Technique," *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1996.

[Jones et al. 97] Michael B. Jones, Daniela Roşu, Marcel-Cătălin Roşu. "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities", In *Proceedings of the 16th ACM Symposium on Operating System Principles*, St-Malo, France, pp. 198-211, Oct. 1997.

[Kaashoek et al. 97] D.R. Engler, M.F. Kaashoek, J. O'Toole Jr., "Application Performance and Flexibility on Exokernel Systems," *Proceedings of the 16th ACM Symposium on Operating System Principles*, Oct. 1997.

[Kay & Lauder 88] J. Kay and P. Lauder, "A Fair Share Scheduler," *Communications of the ACM*, Vol. 31, No. 1, pp. 44-55, 1988.

[Kleinrock 74] L. Kleinrock, *Queueing Systems. Volume 1*, New York: John Wiley, 1974.

[Lampson 68] B.W. Lampson, "A Scheduling Philosophy for Multiprogramming Systems," *Communications of the ACM*, Vol. 10, pp. 613-615, May 1968.

[Liu & Layland 73] C. L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, Vol. 20, No. 1, pp. 46-61, Jan. 1973.

[Nieh & Lam 97] J. Nieh and M. S. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications" *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp. 184-197, Oct. 1997.

[Northcutt 88] J. D. Northcutt, "The Alpha Operating System: Requirements and Rationale" Archons Project Technical Report #88011, Dept. of Computer Science, Carnegie-Mellon, Jan. 1988.

[Ousterhout 82] J.K. Ousterhout, "Scheduling Techniques for Concurrent Systems," *Proceedings of the 3rd International Conference on Distributed Computer Systems*, pp. 22-30, Oct. 1982.

[Ritchie 84] Dennis M. Ritchie, "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Oct. 1984.

[Seltzer & Small 97] M.I. Seltzer, C. Small, "Self-monitoring and Self-adapting Operating Systems," *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pp. 124-129, May 1997.

[Sha et al. 90] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol. 39, No. 9, pp. 1175-1185, Sep. 1990.

[Solomon 98] David A. Solomon. *Inside Windows NT, Second Edition.* Microsoft Press, 1998.

[Waldspurger 95] C.A. Waldspurger, "Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management," Ph.D. dissertation, Massachusetts Institute of Technology, Sep. 1995. Also appears as Technical Report MIT/LCS/TR-667.

RACC: An Approach to Cluster-Based Web Servers

Xiaolan Zhang, Ravi Shanmugan, Michael Barrientos, J. Bradley Chen
Division of Engineering and Applied Sciences
Harvard University

Abstract

RACC is a cluster-based design for scalable cost-effective web servers. Organized around the goal of locality enhancement, the RACC approach seeks to distribute requests arriving at the cluster among the nodes so as to enhance the locality of reference that occurs on individual nodes in the cluster. By improving locality on individual cluster nodes, we can reduce their working set sizes, thereby achieving superior performance for less cost than conventional approaches. We describe here the RACC architecture and its prototype implementation on Windows NT.

1. Introduction

Cluster-based web server designs have great potential because of its scalability and cost-effectiveness. A simple approach to building a cluster-based web server is to put an HTTP request router or "IP sprayer" between the Internet and a cluster of web servers. The router distributes HTTP requests among the cluster nodes, typically assigning clients to server nodes in a round-robin fashion. Adding more nodes to the cluster can increase the aggregate performance of the cluster.

This simple approach to clustering is not a panacea. For example, a server node for a large web site might require a large amount of physical memory in order to handle requests efficiently. Each node added to the system will receive requests for the same set of documents, and so will have the same large memory requirements. If the document set grows beyond the memory size of the server nodes, all nodes will begin to thrash. As a result the server nodes tend to be either expensive, slow, or both.

The RACC design seeks to eliminate this inefficient resource usage by enhancing the inherent locality of the request streams delivered to the server nodes. Rather than distributing requests in a round-robin fashion, it distributes requests based on the URL, such that the current working set of the web server is partitioned among the server nodes. We achieve this improvement by replacing the IP sprayer with a Smart Router in RACC. This design has a number of advantages over the standard IP sprayer approach. First, each node in the cluster is responsible for only a fraction of the total working set of the web server. Second, the size of each node's working set decreases each time a node is added to the cluster. The Smart Router can tune the load presented to each node in the cluster based on that node's capacity. This makes it possible to build the cluster out of relatively inexpensive machines, and to increase the capacity of the cluster by small increments.

2. Implementation

The heart of the RACC cluster is the Smart Router. The Smart Router is partitioned into two layers, the user level High Smart Router (HSR), and the kernel level Low Smart Router (LSR).

The kernel-LSR is a driver sitting above NT's TCP/IP TDI interface. It listens on the HTTP port for a connect request. When a request is received, TCP passes it to the LSR. The LSR extracts the URL from the request and passes it up to HSR. The LSR then waits for the HSR to indicate which cluster node should handle the request. The LSR maintains TCP connections with each cluster node on which it forwards requests. After delivering a request, the LSR ferries data between the two connections until either end closes the connection. A key implementation challenge in the LSR is efficient handling of many simultaneous TCP/IP connections.

The job of the HSR is to monitor the state of the document store, the nodes in the cluster, and properties of the documents passing through the LSR. It then must use the information to make decisions about how to distribute requests over RACC cluster nodes. Internally, the HSR builds a tree structure that matches the document store name-space. Leaves in the tree represent documents that can be requested from the web site. Nodes in the tree represent internal nodes in a hierarchical name space. As the HSR processes requests, it annotates the tree with information about the document store to be applied in load balancing. This information could include document sizes, number of times each document is requested, and compute cycles required to deliver a given document.

We evaluated our design for both web file service and dynamically generated web pages using Lotus Domino. Our preliminary experiments show an improvement in throughput of up to 8x over IP Sprayer for an artificial static web file service workload, and a 20% improvement for a workload based on Lotus Domino.

The Sombrero Distributed Single Address Space Operating System Project

Alan Skousen/Donald Miller
Computer Science and Engineering Department
Arizona State University

The Sombrero operating system project is an experiment to investigate the potential of a distributed large Single Address Space Operating System (SASOS) as a computing environment. We are currently prototyping Sombrero on Alpha 21164 based NT systems. The prototype gives NT the ability to manage 8-terabyte data sets across a network in a single distributed address space. This distributed address space supports a mechanism that allows multiple network users to have access to and a consistent view of the same addresses while at the same time being protected from one another's activities. We also propose changes to current stock processor architecture to facilitate the single large distributed address space design.

Single Address Space Operating Systems (SASOSs) have been researched for a number of years. This research is motivated by the difficulties associated with communication between processes on the same machine and across a network. By reducing the number of namespaces used by an executing program it is possible to provide a commonality of reference that simplifies communication between it and its services. As a result programs and services become smaller and less costly to program and execute more quickly. Because of the paradigm shift new design and programming strategies need to be examined to make full use of the changed architecture.

In Sombrero, protection domains can be used to implement instantiations of Object Oriented Programming base classes that can serve as operating system modules or application programs. Important hardware services are reserved to designated modules and protected by access policy that is enforced by the protection hardware. This allows a non-hierarchical strategy for OS design that permits all services and programs to exist as peers invoking service methods rather than gaining access to resources and services through nested API layers. By this means high speed access directly to low level services using the Sombrero domain switching strategy reduces the cost of a service call to that of a common subroutine.

The Sombrero project is currently being prototyped on two Alpha 21164 boxes with sufficient resources to compile and run Windows NT source code. We have extended the reachable address space by specializing an NT process to include the full range available to the processor. This is done by modifying the NT PALCode to recognize a Sombrero process and to forward TLB misses outside the normal NT range to a Sombrero extension of the PALCode. A pager in NT user space forms the correct TLB entry and handles the miss. With this structure available we extended the PALCode to emulate a protection buffer (RPLB) that is triggered by misses in the TLB. Using this emulated RPLB hardware, individual threads are assigned to separate protection domains within the specialized process. This provides a functional protected single address space that supports the SASOS abstraction we propose. The NT operating system running in Kernel and Executive modes in the NT kernel and NT user space respectively provide the basic I/O facilities to Sombrero, which runs in Supervisor mode.

Thus far the design simulations and performance tests are encouraging. The full implementation of Sombrero, short of the proposed hardware improvements, is under development.

We wish to acknowledge the assistance of Microsoft Corporation for their extensive software support and Digital Equipment Corporation for their technical support.

Alan.Skousen@asu.edu
Donald.Miller@asu.edu
http://www.eas.asu.edu/~sasos

Extending NT Virtual Memory by SCI-based Hardware DSM[1]

Martin Schulz and Hermann Hellwagner
LRR-TUM, Technische Universität München, Germany
{schulzm,hellwagn}@in.tum.de
http://wwwbode.informatik.tu-muenchen.de/Par/arch/smile/sisci/pthreads/

Introduction

Due to their ease of use, shared memory programming models are increasingly becoming of interest for clusters of workstations. The Scalable Coherent Interface (SCI) [1], a new interconnection technology in the SAN area with hardware DSM capabilities, offers a good platform for this programming model allowing a global and transparent SCI Virtual Memory to be built. This forms the basis for transparent shared memory programming libraries, like thread packages [2], which opens the architecture of clusters of workstations to a second programming model (next to message passing) and with this to a new group of applications and users.

SCI Virtual Memory

The DSM offered by SCI is based only on sharing contiguous physical memory segments. In order to construct a global virtual memory, techniques well known from software DSM packages need to be applied. The data is distributed at page granularity and the SCI-VM software layer provides applications with a consistent view onto the distributed memory resources. Unlike in software DSM systems, however, remote pages need not be replicated or migrated; they can simply be mapped by the SCI adapter and accessed using SCI's remote memory capabilities.

The SCI-VM concept requires the ability of mapping individual physical pages into the virtual address space. Unfortunately, Windows NT does not provide or document this functionality. Therefore, the only way for implementation is through bypassing the operating system and mapping single pages by directly manipulating the CPU's page table.

Implementing SCI-VM and first results

Using the techniques described above, we have implemented a restricted test version of the SCI-VM on a reduced cluster consisting of two PCs (233 MHz Pentium II with 440FX Chipset). One node was used for computation while the second acted as a memory server only supplying remote memory. This memory is interleaved with the local memory on the computation node at page granularity in a round robin fashion.

On top of this global memory we tested two synthetic kernels, a sum and a matrix multiplication code as well as one larger application, the "volrend" code from the SPLASH-2 suite [3]. All codes ran in three different versions: exclusively on local memory to provide a baseline comparison, in an unoptimized version, and in an optimized version where we used hardware buffering and prefetching techniques from the SCI adapter card as well as caching. The latter optimization is not directly supported by the SCI hardware as it does not implement cache consistency. To compensate, a software cache coherence scheme was applied.

As expected, in the unoptimized case all codes performed several orders of magnitude slower than the baseline comparison due to the long read latencies over the SCI network (≈ 6 μs). With optimizations activated, however, all codes performed very well yielding an overhead of less than 75%. In some cases, an overhead as low as 5.1% could be obtained. These numbers indicate that these applications, when run in parallel on several compute nodes, would exhibit an acceptable speedup. Future experiments with several compute nodes will provide actual speedup numbers.

References

[1] IEEE Computer Society. *IEEE Std 1596-1992, IEEE Standard for the Scalable Coherent Interface*, Aug. 1993.

[2] M. Schulz. *SISCI Pthreads: SMP-like Programming on an SCI-cluster*, HPCN Europe 1998.

[3] S. Woo, M. Ohara, E. Torrie, J. Jaswinder, and A. Gupta. *The SPLASH-2 Programs: Characterization and Methodological Considerations*, ISCA 1995.

[1] This work is supported by the European Commission in the Fourth Framework Programme under ESPRIT HPCN Project EP 23174: Standard Software Infrastructure for SCI-based Parallel Systems (SISCI)

Chime: A Windows NT based parallel processing system[1]

Shantanu Sardesai
Tandem Computers Incorporated
shantanu.sardesai@tandem.com

Partha Dasgupta
Arizona State University
partha@asu.edu

1. Introduction

Shared memory multiprocessors are the best platform for writing parallel programs. These platforms support a variety of parallel processing languages (such as CC++ which provide programmer-friendly constructs for expressing shared data, parallelism, synchronization and so on. However the cost and lack of scalability and upgradability of shared memory multiprocessor machines, make them a less than perfect platform.

Distributed Shared Memory (DSM) has been promoted as the solution that makes a network of computers look like a shared memory machine. This approach is supposedly more natural than the message passing method used in PVM and MPI.

However, most programmers find this is not the case. The shared memory in DSM systems do not have the same access and sharing semantics as shared memory in shared memory multi-processors. For example, only a designated part of the process address space is shared, linguistic notions of global and local variables do not work intuitively, parallel functions cannot be nested and so on.

Chime is the *first* system that provides a true shared memory multiprocessor environment on a network of machines. It achieves this by implementing the CC++ language (shared memory) on a distributed system. In addition to shared memory, parallelism and synchronization features of CC++, Chime also provides fault-tolerance and load balancing.

2. Chime Features

Chime addresses most of the problems in a simple, clean and efficient manner by providing a multiprocessor-like shared memory programming model on network of workstations, along with automatic fault-tolerance and load balancing. Some of the salient features of the Chime system are:

1. Complete implementation of the shared memory part of the CC++ language.
2. Support for nested parallelism; i.e. a parallel task can spawn more parallel tasks.
3. Consistent memory model, i.e. the global memory is shared and all descendants (which execute in parallel) share the local memory of a parent task.
4. Machines may join the computation at any point in time (speeding up the computation) or leave or crash at any point without affecting the progress (slowdowns will occur).
5. Faster machines do more work than slower machines, and the load of the machines can be varied dynamically (load balancing).

In fact, there is very little overhead associated with these features, over the cost of providing DSM. This is a documented feature that Chime shares with its predecessor Calypso.

3. Chime Architecture

A program written in CC+ is preprocessed to convert it to C++. Then it is linked with the Chime runtime library and a single executable file is generated. This executable is executed on a network of machines (or workstations). One of the workstations is designated as the *manager* and the rest as *workers*. All run the same executable.

A program starts on the manager. When the program reaches a parallel construct, parallel tasks are generated and are allocated by the manager to the waiting workers. During the execution of the parallel step, the manager does scheduling and allocation of parallel tasks, as well as memory management.

The manager and worker are multithreaded, the programmer written code is executed by one thread and the other thread handles all the runtime functions. The runtime functions include servicing DSM requests, inter-task synchronization, cactus stacks for proper stack sharing and scheduling that handles fault tolerance and load balancing. Chime is implemented on Windows NT.

The performance tests of Chine show that it it performs well for a variety of parallel programs. The nested parallelism and synchronization support however adds considerable overhead in a distributed system. The Chime system can be downloaded from http://milan.eas.asu.edu.

[1] This research is partially supported by grants from DARPA/Rome Labs, Intel Corporation and NSF.

Distributed Preemptive Scheduling on Windows NT[1]

Donald McLaughlin and Partha Dasgupta
Arizona State University
partha@asu.edu

1. Introduction

All multitasking operating systems use preemptive scheduling. Many multiprocessor systems also employ preemptive inter-task scheduling when they run parallel computations. However, preemptive scheduling in distributed systems is rare, if not non-existent.

Consider a cluster of workstations, running a *parallel* application. The application divides itself into a set of tasks. The scheduler assigns these tasks to a set of workstations. Often the tasks are not of equal length, the machines are not of equal speeds and tasks can create further subtasks. These situations lead to non-optimal matches of workers to tasks causing executions that do not complete as quickly as it would be possible in a better matched case. Also, the granularities of the tasks may be small, leading to high overhead.

2. Distributed Scheduling

Our research has addressed such problems in a variety of ways. We have developed scheduling algorithms, both non-preemptive and preemptive that provide good throughputs in managing distributed computations, even when the granularities of tasks are small.

Our research environment consists of the *Chime* parallel processing systems running on the Windows NT operating system. This system support parallel processing on a network of workstations, with support for *Distributed Shared Memory (DSM), fault tolerance, adaptive parallelism* and *load balancing*. The default scheduler used in Chime is *Eager Scheduling*. Eager Scheduling is similar to a FIFO scheduling algorithm augmented to provide fault tolerance (by assigning uncompleted tasks repeatedly).

Hence, without intelligent scheduling, the faster machines idle at barrier points waiting for the slower machines to finish, causing reductions in throughput. In addition, small mismatches in the number of machines and tasks cause large idle times and low granularities cause high overhead.

We have found that various preemptive scheduling algorithms can be used in such situations for significant performance improvements, in spite of the overhead of preemptive scheduling in distributed systems.

3. Preemptive Scheduling

Over the last few years we have simulated and implemented a host of preemptive scheduling algorithms. We now present two such algorithms.

The first algorithm is a variation of the well-known round robin algorithm. We call this the *Distributed, Fault-tolerant Round Robin* algorithm. In this algorithm, a set of n tasks is scheduled on m machines, where n is larger than m. Initially, the first m tasks are assigned to the m machines. Then, after a specified amount of time (time quantum), all tasks are preempted and the next m tasks are assigned. This continues in a circular fashion until all tasks are completed.

The second is the *Preemptive Task Bunching* algorithm. All n tasks are bunched into m bunches and assigned to the m machines. When a machine finishes its assigned bunch, all the tasks on all other machines are preempted and all the remaining tasks are collected and re-bunched (into m sets) and assigned again. This algorithm works well for both large-grained and fine-grained tasks even when machine speeds and task lengths vary.

4. Implementation and Performance

We have implemented the algorithms on the Chime parallel processing system running on Windows NT. The major roadblock turned out to be process migration under NT. The lack of signals posed the greatest problem as a thread can only be interrupted by another thread that suspends it. Care has to be taken to ensure that the thread to be migrated is not suspended waiting for a runtime event. Race conditions and starvation conditions have been encountered.

The final system runs well, and performance results are very encouraging. We found that the round-robin scheduler provided acceptable performance on large grained programs, but was hampered by the migration overhead. The task bunching scheduler performed really well in a wide variety of situations. More information and papers can be found at http://milan.eas.asu.edu.

[1] This research is partially supported by grants from DARPA/Rome Labs, Intel Corporation and NSF.

SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit

Ashvin Goel, David Steere, Calton Pu, Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute, Portland

Currently, the task of building adaptive system software relies on wizardry. Feedback controls for such software systems are written in an ad-hoc manner and are often brittle. As a result, it is difficult to move an existing control, such as TCP flow control [1], to a new domain such as admission control for CPU scheduling. In addition, existing controls are built with implicit assumptions about the system's run-time environment and can become unstable in the face of large or discontinuous variations in the environment.

We advocate a systematic approach for building adaptive system software based on feedback-control theory. We have implemented the SWiFT toolkit that incorporates this approach. Feedback control helps produce predictable control components. It requires the control goal and design specifications to be clearly stated, thus allowing analysis of properties such as stability. Our approach allows us to leverage the existing body of knowledge in hardware control for controlling software systems.

SWiFT allows systematic implementation of feedback-control mechanisms by providing a framework and methodology for building controls that are modular, dynamically reconfigurable, and predictable. Modularity results from our use of components and containers as the underlying abstraction. SWiFT also allows easy dynamic reconfiguration of components by limiting the interaction between components to a simple input/output model and by supporting guarding and replugging of controllers [2]. This reconfiguration allows the application to adapt efficiently across a wide range of operating conditions. SWiFT supports predictability by providing analysis tools based on control theory. Also SWiFT provides GUI debugging tools such as a software oscilloscope and a library of feedback components such as low pass filters to simplify building adaptive system software.

We have implemented SWiFT in C++ and Java and we have applied it to user-level applications running on Windows NT. Version 1.0 of SWiFT is available (along with a tutorial) at http://www.cse.ogi.edu/DISC/projects/swift. We are currently building a visual editor for designing, implementing, and monitoring controls using SWiFT.

The basic abstractions in SWiFT are *feedback components* and f*eedback containers*. Feedback components read data from their *input port*(s), calculate an output value based on their characteristic behavior, and pass the value to their *output port.* A control circuit is built by connecting a component's output port to input ports of one or more components. A feedback container, provide modularity and hierarchical structure. It is a feedback component that contain other feedback components and containers, and defines a circuit of connections among its children and its input and output ports.

Another aspect of SWiFT, *dynamic reconfiguration*, consists of tuning a control circuit's parameters, or replacing parts of the control circuit at run-time. This reconfiguration is the *controlled* system's response to drastic changes in the underlying environment that violate the controller's basic design assumptions. For instance, TCP's adaptive flow-control algorithm performs poorly over wireless links. Replacing this policy with one more suited to wireless use results in better performance [3]. Dynamic reconfiguration is similar to hardware hot-swapping with the addition that the swapping is done automatically. SWiFT can dynamically reconfigure feedback components in controllers based on user-specified predicates on system properties, called *guards*. Guards are explicitly programmed by the controller's designer.

We are currently developing adaptive control mechanisms using SWiFT on three diverse system domains on NT: a streaming media player, an informed multimedia prefetching system, and a feedback-based proportional share CPU scheduler.We have chosen NT for its modular structure, and widespread use as a platform for multimedia applications. Our designs make heavy use of NT's user-level device drivers and Microsoft's DirectShow infrastructure.

This project was supported in part by DARPA contracts/grants N66001-97-C-8522, N66001-97-C-8523, and F19628-95-C-0193, and by Tektronix, Inc. and Intel Corporation.

References

[1] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[2] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *SOSP'95*, pages 314–324, Copper Mountain resort, Colorado, USA, December 1995.

[3] R. Yavatkar and N. Bhagwat. Improving end-to-end performance of TCP over mobile internetworks. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, U.S., 1994.

COM on a Multicast Transport

P. Emerald Chung and **Yennun Huang**
{emerald, yen}@research.bell-labs.com
Bell Laboratories, Lucent Technologies
Murray Hill, New Jersey

Yi-Min Wang
ymwang@microsoft.com
Microsoft Research
Redmond, Washington

Introduction

Due to the wide availability of the Internet and intranets, the use of multicast applications, such as document sharing, collaborative groupware and multimedia delivery, has grown quickly. Another trend in software development is rapid application development by using off-the-shelf components. Microsoft Component Object Model (COM) [COM 95] has become a popular paradigm for fast component integration. In this paper, we extend the COM object model to integrate with a multicast transport protocol, called RMTP [Paul 96]. The goal is to develop a framework that enables building multicast applications from off-the-shelf components. We demonstrate our initial results with a stock quotes dissemination system.

COM over RMTP

RMTP is built upon the foundation of IP multicast, and provides sequenced, lossless delivery of a data stream from one sender to a group of receivers. There is little correlation among the group of RMTP receivers. The Internet Group Management Protocol (IGMP) establishes a way of joining and leaving multicast groups. RMTP has been used commercially for data dissemination in wide area network where strong state consistency among the receivers is not required.

COM specifies an architecture, a binary standard, and a supporting infrastructure for building, using, and evolving component-based applications. It extends the benefits of object-oriented programming such as encapsulation, polymorphism, and software reuse to a dynamic and cross-process setting.

The overall architecture of COM+RMTP is shown in the figure. We leverage many mechanisms from the *Distributed COM (DCOM)* [Brown 96] technologies. DCOM is an object-oriented RPC protocol. It provides the infrastructure that allows a client process to create server objects on remote machines and to invoke their methods in a location-transparent way. While the standard DCOM implementation is designed for one-to-one, synchronous invocation, RMTP is used for one-to-many, asynchronous invocation. By encapsulating such semantic difference in the channel and stub manager components (see figure), we are able to implement our system by exploiting the DCOM custom marshaling mechanism to replace the RPC transport with the RMTP protocol, while reusing standard proxy/stub components provided by DCOM for data marshaling.

An Application

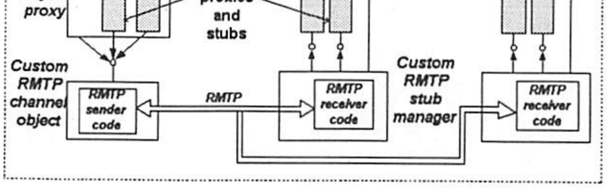
We use the COM/RMTP integration to implement a stock quote dissemination system. A client gets the stock quote from the Internet and makes COM method calls to multicast the data to a display component running on many (server) machines. The client and the servers communicate through a well-defined COM interface. It is interesting to note that, as we added more types of data to be delivered, new client and a new interface in the server were built to accommodate the changes without breaking old clients.

Research Issues

There are several issues to be addressed in order to make this framework practical: (1) a protocol to advertise a group and its COM interfaces, (2) a mechanism to handle multiple callbacks if servers need to notify the client, and (3) a tool to automate the development of multicast-aware components. We are currently working on these issues and exploring potential applications.

References:

[Paul 96] J. C. Lin and S. Paul , "RMTP: A Reliable Multicast Transport Protocol", INFOCOM '96, pp. 1414-1424.
[COM 95] The Component Object Model Specification, http://www.microsoft.com/oledev/olecom/title.htm
[Brown 96] N. Brown, C. Kindel, Distributed Component Object Model Protocol -- DCOM/1.0, http://premium.microsoft.com/msdn/library/techart/msdn_dcomprot.htm



NOTES

NOTES

NOTES

NOTES

NOTES

NOTES

NOTES